18.405 — Advanced Algorithms

Class by David Karger

Notes by Sanjana Das

Fall 2024

Lecture notes from the MIT class **18.415** (Advanced Algorithms), taught by David Karger. All errors are my own.

Contents

1 September 4, 2024 **9**
- 1.1 Fibonacci heaps . . . 9
- 1.2 Motivation . . . 9
- 1.3 Fibonacci heaps . . . 10

2 September 6, 2024 **14**
- 2.1 Review . . . 14
- 2.2 Decrease-key . . . 16
- 2.3 Analysis . . . 17
  - 2.3.1 Cascading cuts . . . 18
  - 2.3.2 The exponential-in-degree claim . . . 18
- 2.4 A coda . . . 19
- 2.5 Back to MSTs . . . 20

3 September 9, 2024 **22**
- 3.1 Persistent data structures . . . 22
  - 3.1.1 A first attempt — the fat nodes method . . . 23
  - 3.1.2 A second attempt — path copying . . . 24
  - 3.1.3 The solution — lazy path copying . . . 24
- 3.2 Logistics . . . 26
- 3.3 An application: planar point location . . . 26

4 September 11, 2024 **28**
- 4.1 Binary search trees . . . 29
- 4.2 Some motivation . . . 29
- 4.3 The splay operation . . . 30
- 4.4 The analysis . . . 32
- 4.5 The final and initial potential . . . 36
- 4.6 Some more applications of the access lemma . . . 37
  - 4.6.1 Static optimality . . . 37
  - 4.6.2 Fingering . . . 38
  - 4.6.3 Caches . . . 38

4.6.4 Final comments . . . 38

5 September 13, 2024 **39**
5.1 More about splay trees . . . 39
5.1.1 Insert and delete in splay trees . . . 39
5.1.2 Split and merge . . . 39
5.1.3 Some imperfections of splay trees . . . 40
5.1.4 Compression . . . 41
5.1.5 Dynamic optimality conjecture . . . 41
5.2 Introduction to indirect addressing . . . 41
5.3 Motivation: shortest paths . . . 42
5.3.1 Dial's algorithm . . . 42
5.3.2 Conserving space . . . 43
5.4 2-level buckets . . . 43
5.5 Tries . . . 44
5.6 Laziness . . . 45

6 September 16, 2024 **47**
6.1 Tries . . . 47
6.2 Van Emde Boas priority queues . . . 49
6.2.1 Inserts . . . 50
6.2.2 Delete-min . . . 50
6.2.3 Limitations and issues . . . 51
6.3 Hashing . . . 52
6.3.1 Motivation . . . 52
6.3.2 The problem . . . 52
6.3.3 Hash families . . . 53
6.3.4 Random hash functions . . . 53
6.3.5 Carter–Wegman universal hashing . . . 54

7 September 18, 2024 **55**
7.1 A pairwise independent hash family . . . 55
7.2 Conclusions . . . 56
7.3 Motivation for perfect hashing . . . 57
7.4 A starting point — perfect hashing with quadratic space . . . 57
7.5 Decreasing space — a first attempt . . . 58
7.6 Perfect hashing — the solution . . . 59
7.7 Dynamic perfect hash tables . . . 60
7.8 Intro to combinatorial optimization . . . 60
7.8.1 A paradigm . . . 61
7.9 The maximum flow problem . . . 61

8 September 23, 2024 **61**
8.1 History of max-flow . . . 61
8.2 Definition of flows . . . 62
8.3 A recipe for combinatorial optimization . . . 63
8.4 Understanding flows . . . 64
8.5 Verification of max-flows . . . 66
8.5.1 Augmenting paths . . . 66
8.6 Algorithms . . . 69
8.6.1 Number of augmentations . . . 69

9 September 25, 2024 **70**
9.1 Recap . . . 70
9.2 Augmenting paths algorithms . . . 70
9.2.1 Rational capacities . . . 71
9.2.2 Real capacities . . . 71
9.3 A greedy improvement . . . 72
9.3.1 Finding a max-capacity augmenting path . . . 72
9.3.2 How many iterations? . . . 73
9.3.3 Integer capacities . . . 73
9.3.4 Rational capacities . . . 74
9.4 Scaling algorithm for max flow . . . 74
9.5 Strongly polynomial max-flow . . . 76
9.6 Shortest augmenting path algorithm . . . 77

10 September 30, 2024 **79**
10.1 Review . . . 79
10.2 Motivation . . . 79
10.3 The blocking flow technique . . . 80
10.4 Finding a blocking flow . . . 81
10.4.1 The unit capacity case . . . 81
10.4.2 General capacities . . . 83
10.4.3 Scaling blocking flows . . . 84
10.5 A strongly polynomial algorithm . . . 84
10.6 Link-cut trees . . . 86
10.7 Concluding remarks . . . 86

11 October 2, 2024 **88**
11.1 Min-cost flow . . . 88
11.2 Min-cost circulation . . . 88
11.3 Deciding optimality . . . 90
11.4 Algorithm for min-cost flow . . . 91
11.5 Price functions . . . 92

12 October 7, 2024 **95**
12.1 Review . . . 95
12.2 A greedy approach . . . 96
12.3 A shortest augmenting paths algorithm . . . 98
12.4 Dealing with negative cycles . . . 98
12.5 Scaling min-cost circulation . . . 99
12.6 Complementary slackness . . . 100
12.7 Conclusion . . . 102

13 October 9, 2024 **102**
13.1 Linear programming . . . 102
13.2 Some notation . . . 103
13.2.1 Canonical form . . . 103
13.2.2 Standard form . . . 104
13.3 An overview . . . 105
13.4 The linear equality case . . . 105
13.4.1 Sizes of numbers . . . 106
13.4.2 Showing infeasibility . . . 107

13.4.3 Duality . . . 108
13.5 Some geometric intuition . . . 109
13.6 Corners . . . 109

14 October 11, 2024 **111**
14.1 Review . . . 111
14.2 Characterizing the optima . . . 112
14.3 Size of the optimum . . . 114
14.4 A first algorithm . . . 114
14.5 Verifying optima . . . 115
14.6 Taking linear combinations . . . 115
14.7 The dual LP and weak duality . . . 116
14.8 Strong duality . . . 117
14.8.1 A physics proof . . . 118

15 October 16, 2024 **119**
15.1 Recap of the physics proof . . . 119
15.1.1 Complementary slackness . . . 119
15.2 Formal proof of strong duality . . . 120
15.3 Consequences . . . 122
15.4 Rules for taking duals . . . 123
15.5 Dual of shortest-paths . . . 124

16 October 18, 2024 **125**
16.1 Review . . . 125
16.2 Dual of maximum flow . . . 126
16.2.1 Interpreting the LP . . . 129
16.2.2 Complementary slackness . . . 130
16.2.3 Proof of max-flow min-cut . . . 131

17 October 21, 2024 **132**
17.1 Review . . . 132
17.2 Min-cost circulation . . . 132
17.3 Algorithms for linear programming . . . 134
17.4 The simplex algorithm . . . 135
17.4.1 Characterizing vertices . . . 135
17.4.2 The simplex algorithm . . . 136
17.4.3 Potential issues . . . 138
17.4.4 Runtime of simplex . . . 139

18 October 23, 2024 **140**
18.1 The simplex algorithm and duality . . . 140
18.2 The ellipsoid algorithm . . . 141
18.3 Hunting lions . . . 141
18.4 Ellipsoids . . . 142
18.5 Finding a smaller ellipsoid . . . 142
18.6 Starting ellipsoid size . . . 145
18.7 Ending ellipsoid size . . . 145
18.8 Runtime . . . 146
18.9 Separation oracles . . . 146
18.10 Interior point . . . 147
18.11 Physics motivation . . . 147

18.12A potential function . . . 148
18.13The central path . . . 148

19 October 28, 2024 **149**
19.1 Dealing with NP-hard problems . . . 149
19.2 Setup . . . 149
19.2.1 Optimization problems . . . 149
19.2.2 NP-hardness . . . 150
19.3 Absolute approximation algorithms . . . 150
19.3.1 Planar graph coloring . . . 150
19.3.2 The scaling issue . . . 151
19.4 Relative approximation . . . 153
19.5 Proving something is an α-approximation . . . 153
19.6 Greedy algorithms . . . 153
19.7 Max-Cut . . . 154
19.8 Set cover . . . 154
19.9 Vertex cover . . . 155
19.10Scheduling theory . . . 156

20 October 30, 2024 **157**
20.1 Scheduling theory . . . 158
20.2 The best approximation ratios . . . 159
20.3 Approximation schemes . . . 159
20.4 Knapsack . . . 160
20.4.1 Dynamic programming for small profits . . . 160
20.4.2 Handling large numbers . . . 161
20.4.3 Improving the runtime . . . 162
20.5 FPASs and pseudopolynomial algorithms . . . 163
20.6 Enumeration . . . 164

21 November 1, 2024 **165**
21.1 Scheduling and enumeration . . . 165
21.1.1 Constant number of machines . . . 165
21.1.2 Arbitrary numbers of machines . . . 165
21.1.3 A special case — k distinct sizes of jobs . . . 166
21.2 Geometric rounding for job sizes . . . 166
21.2.1 The algorithm . . . 167
21.3 General comments . . . 168
21.4 Relaxation and the travelling salesman problem . . . 168
21.4.1 General relaxation . . . 169
21.4.2 Relaxation for Metric TSP . . . 169
21.4.3 The Christofides heuristic . . . 170
21.4.4 Further work . . . 171

22 November 4, 2024 **172**
22.1 Relaxation overview . . . 172
22.2 A scheduling problem — $1 \mid r_j \mid \sum c_j$ . . . 172
22.2.1 Relaxation by preemption . . . 173
22.2.2 Rounding . . . 174
22.2.3 Improvements . . . 175

22.3 ILP relaxation — vertex cover . . . 175
22.3.1 Improvements . . . 176
22.4 Logistics . . . 177
22.5 Facility location . . . 177
22.5.1 An integer linear program . . . 178
22.5.2 Ideas for rounding . . . 178
22.5.3 Filtering . . . 179
22.5.4 Facility opening . . . 180

23 November 6, 2024 **180**
23.1 Facility location . . . 180
23.1.1 Review . . . 180
23.1.2 Facility opening . . . 181
23.1.3 The approximation ratio . . . 182
23.1.4 Can we do better? . . . 183
23.2 Max-SAT and randomized rounding . . . 183
23.2.1 A random solution . . . 183
23.2.2 An integer linear program . . . 183
23.2.3 Randomized rounding . . . 184
23.2.4 The distribution of z_j . . . 184
23.2.5 Combining the two approximations . . . 186
23.3 Fixed-parameter tractability . . . 186
23.4 Vertex-Cover and the bounded search tree method . . . 187
23.5 The kernel method . . . 187
23.6 Fixed-parameter tractability theory . . . 188

24 November 8, 2024 **188**
24.1 Online algorithms . . . 188
24.2 Setup and definitions . . . 189
24.3 Ski rental . . . 190
24.4 Market . . . 191
24.5 $P \parallel C_{\max}$ online . . . 193
24.6 Paging . . . 194

25 November 13, 2024 **196**
25.1 Paging . . . 196
25.1.1 Proving $(k+1)$-competitivity . . . 196
25.1.2 Improving $k+1$ to k . . . 197
25.1.3 Tightness of k-competitivity . . . 197
25.1.4 Final remarks . . . 197
25.2 Randomized competitive algorithms . . . 198
25.3 Types of adversaries . . . 198
25.4 Marking algorithm for paging . . . 199
25.4.1 Can we do better? . . . 202

26 November 15, 2024 **202**
26.1 Review . . . 202
26.2 Game theory . . . 202
26.3 Optimal strategies . . . 203
26.4 Yao's minimax principle . . . 204
26.5 A lower bound for randomized paging . . . 206

26.6 The k-server problem . . . 207
26.6.1 First attempt — greedy . . . 208
26.6.2 Harmonic algorithm . . . 209
26.6.3 k-server on a line . . . 209

27 November 18, 2024 **210**
27.1 The k-server problem . . . 210
27.1.1 The double coverage algorithm . . . 210
27.1.2 The distance to OPT . . . 211
27.1.3 A potential function for amortization . . . 212
27.1.4 The analysis . . . 212
27.1.5 Improvements . . . 214
27.2 Computational geometry . . . 214
27.3 Orthogonal range searching . . . 215
27.3.1 The 1D version . . . 215
27.3.2 Generalizing to 2D . . . 216
27.3.3 Improving construction time . . . 216

28 November 20, 2024 **217**
28.1 Dynamic orthogonal range query . . . 218
28.1.1 A first attempt . . . 218
28.1.2 Treaps . . . 218
28.2 Sweep algorithms . . . 219
28.3 Convex hull . . . 219
28.3.1 Sweep line algorithm . . . 219
28.3.2 Runtime . . . 220
28.4 Halfspace intersections . . . 220
28.5 Segment intersections . . . 221
28.6 Voronoi diagrams . . . 222
28.7 Post office problem . . . 222
28.8 Structure of Voronoi cells . . . 222
28.9 Answering nearest neighbor queries . . . 224
28.9.1 Space complexity . . . 224

29 November 22, 2024 **226**
29.1 Intro – constructing the Voronoi diagram . . . 226
29.2 Sweep line approach . . . 226
29.3 The beach line . . . 226
29.4 Descent of parabolas . . . 227
29.5 Site events . . . 228
29.6 Circle events . . . 228
29.7 Ruling out other events . . . 228
29.8 Data structures . . . 229
29.8.1 Circle events . . . 230
29.9 Time analysis . . . 230
29.10Higher dimension . . . 230
29.11Delaunay triangulations . . . 231
29.12External memory algorithms — the model . . . 231
29.13Basic operations . . . 232
29.14Matrices . . . 232

30 November 25, 2024 **234**
30.1 Linked list data structures . . . 234
30.2 Search trees . . . 235
30.3 Sorting . . . 237
30.4 More on sorting and searching . . . 239
30.5 Cache-oblivious algorithms . . . 240
30.5.1 Scanning an array . . . 240
30.5.2 Divide and conquer — a general approach . . . 240
30.5.3 Matrix multiplication . . . 241
30.5.4 Binary search trees . . . 241

31 December 2, 2024 **242**
31.1 Parallel algorithms . . . 242
31.2 Models . . . 242
31.3 The circuit model . . . 243
31.4 An example — addition . . . 245
31.5 The PRAM model . . . 248
31.6 Computing max . . . 249

32 December 4, 2024 **251**
32.1 Circuits vs. PRAM . . . 251
32.2 List ranking . . . 253
32.3 Parallel search . . . 255
32.4 Parallel sorting . . . 256

§1 September 4, 2024

The first unit of the course is data structures.

§1.1 Fibonacci heaps

Data structures are ojbects that maintain state as you perform various read/write operations on them. We want the time it takes for each operations to be small.

§1.2 Motivation

To motivate the fibonacci heap data structure, we'll talk a bit about shortest paths. David assumes we're familiar with 6.1220/6.046, and this will be taken for granted. So we've seen shortest paths and minimum spanning trees.

There are more of them; so what algorithms have we seen for MST? One is Kruskal. This works by contracting edges in increasing order. Another is Prim. You expand components one edge by one edge, each time choosing the minimal — you grow a tree by repeatedly adding the 'closest' vertex to the tree. This is analogous to Djikstra's shortest paths algorithm. The only difference is your definition of close.

The third, more unusual, MST algorithm is Bourka's algorithm, which is kind of a parallelized Kruskal's algorthm (or parallelized Prim). You're kind of growing all over the graph at the same time and using local information about what's int eh MST to get lots of edges at once. WE'll talk about all three at various times.

For now we'll focus on Prim. Here we need to keep finding the closest vertex. This suggests having a data structure for maintaining the closest vertex.

What operations do we need to support in that data structure? We need an `extract-min` operation. We also need `insert`. And we need `decrease-key` to update distances (we only decrease, never increase). (We also want `find-min` to look up the min withou textract, but that's trivial). There's also `init` (creating the data structure to start with).

What generically do we use for a data structure that uspports these — the fully generic name is *priority queue*; most priority queues we use go under the name of *heap*. The most commonly known is the standard binary heap. This supports all these operations in — extract-min, insert, decrease-key are all $O(\log n)$ time. So if we run them on a graph with m edges and n vertices, we get runtime $O((m+n)\log n)$. And m and n are always bound to edges and vertices throughout this class.

That's not bad; it's pretty close to linear — but it's not linear, so we can hope for improvements. One is by trading off the cost of operations. How many of these operations do we need for shortest-paths or Prim? How many times do we extract the min — this is n times. We do an insert n times. And we decrease-key m times. So there's lots more decrease-keys; and so we might hope to speed up the decreases even if it costs us in the others.

One way we can do this is by doing a d-heap, which is like a binary heap but with degree d. ALl the operations carry over in the obvious way. Then how long does insert take in a d-heap — this is $\log_d n$. Decrease-key is also $\log_d n$. But what this does to extract-min is — you have to promote one of the children, but now there's d you have to look at, and you have to look at its children to promote one of them; there's $\log_d n$ levels but you do d work at each, so this tkaes you $d \log_d n$ time. If you put these together, the overall runtime for Prim or shortest-paths you get is n inserts and extract-mins is $nd \log_d n$, and m decrease-keys is $m \log_d n$. So you get $nd \log_d n + m \log_d n$.

But d is completely under our control, so the question is, what d gives us a good bound? To get the answer, we want to minimize this runtime; if this were a calculus course you would take derivatives but that's too

much work. We only care about asymptotics, so we use the fact one is increasing and one is decreasing, so the point at which they meet is going to be a lower bound on the minimum sum. On the other hand, if we use this point ,the sum is only twic ethis value. So the answer is somewhere between this value and twice this value, and asymptotically those are the same. So we just set $nd \log_d n = m \log_d n$, which means we want $d = \frac{m}{n}$. This gives a runtime of $m \log_{m/n} n$. This is a slight but significant improvement on $m \log_2 n$ — in particular, if you have a *dense* graph (with $m = n^2$), then m/n is n and $\log_n n$ is a constant, so your runtime i slinear. So this gives you an algorithm which is already linear time on dense graphs.

This is really just an aside; we're going to instead of tradeoffs, find a smart data structure that just does better without slowing down.

§1.3 Fibonacci heaps

This is a data structure that aims to improve on the basic time bounds. They were developed by Fredman and Tarjan in 1987. YOu will become very familiar with the name Tarjan; he is the god of data structures, and eveyr incredible data structure has his name on it somewhere.

The key principle we will use this data structure to demonstrate is laziness, which is a very important concept for MIT students to learn. There is an aspect all of us have learned this well — true laziness involves never working, but the point of a data structure is to work, so it has to work. But certainly you should never work until you have to — you work at the last minute. (This relates to the 'should you grade early' question.)

Another equally important facet is that when you are forced to work, you should make your work count — if you must work, you should use the work to decrease future work. In particular, in data structures this means you find ways to simplify the data structure, to make future work easier while you're doing the current work.

Another way to look at this is an adversarial model. If we think of the person usign the data structure as a nasty person forcing us to work (that would be David in this class), you want to force the adversary to do a lot of work to make you do a lot of work. In data structures applicaitons, the goal is that — you want a data structure where if it gets complicated enough that you have to do lots of work, that only hpepened because the adversary did lots of work making it complicated. This will allow us to *amortize* the data structure work against the amount of work of the adversary. I have to do lots of work, but spreading all that work out over all the different operations the adversary asked me to do means on any particular one I wasn't really doing a lot of work.

When you amortize, there are lots of problem-specific ways to do it. But we are going to demonstrate *potential functions* to account for the work. The idea of a potential function is it measures the maount of deferred work int he data structure. The adversary's work is going to increase the potential; we will charge the data structure's work against that potential. And that allows us to transitively relate the data structure work to the number of operations requested by the adversary.

This is an analytic technique we're going to use to analylze the performance of o ur data structure — we will do an amortized analysis where we want to say every X operation takes Y time on average. The way we'll do this is by defining a potential function, saying each operation adds — potential to the function, and when we do work it removes potential from teh potential function. WE'll design it to always b epositive so you can't remove an infintie amount of work, so we can't be forced to do too much work because the structure woul drun out of potential.

For example, if David tells you every time you insert something into the structure you have to climb another flight of stairs in Stata, and for deletion I only do the work coming from you descending. Then I can't do more deletion work than the number of floors you've gone up — I can't be forced to do an infinite amount of work because each work I do takes you fruther down int eh Stata center. (WE'll see an example very soon. But this is a completely general technique that applies to lots of data structures.)

Lastly as a teaser for why the name Fibonacci heaps:

let's come back to the lazy approach in the context of heaps. I want to be as lazy as possible in implementing a heap.

Let's start with a heap that just has init and insert operations. Init is obviously very quick. What's the laziest possible way to respond to an insert request? You can just append the value to a list. That's too much work; the laziest thing you can do is to just say 'okay' and do nothing. Then both are $O(1)$.

Okay, but we may want to support some other operation, We do want to support delete-min as well. So ew can't just drop the item on teh floor; we have to save it somewhere. The obvious way to do this is to add it to an array or linked list (wel'l use linked lists because we'll manipulate this later on).

Now we have an insert that actually can support delete-min. But then on a delete-in you have to scan for the minimum, so if n itesm ahve been inserted, this will cost us $O(n)$ time to do delete-min. Is this good or bad? That depends on how you measure — it's n times the one operation. But if I insert n imtes but do just one delete-min, that's fine — in order to force me to do n work on deletion you had to do n insertions. So it's n work total, but only $O(1)$ per operation, if we only do one deletion. The weakness in this is what happens on the next deletion — we have to scan again. So it's fine for the first delete-min, but if we have many delete-mins, it's too expensive; then it ends up as $O(n)$ per operation, rather than $O(1)$ per operation.

So, what do we do about this? WEll, one thing you do wiht traditional binary heaps is do a lot of work during insertions to keep delet-emin work small. But that doesn't help our procrastination vision.

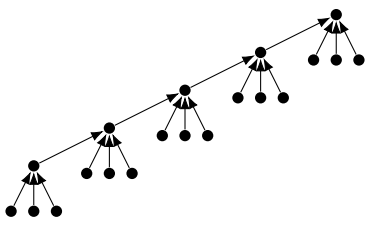
So if we're onily going to work when we're forced to at delete-time — well we knwo we're wokring at this time, so let's use the work we're doing to simplify the data structure to make things better fo rthe next operation.

If we're scanning through teh data structure, what else can we do? You could sort the list, but that's a lot of work — scanning takes us linear time, but sorting would take $n \log n$ time. So that's working ahead of time.

But if we've just done this, you learn various other relationships from doing comparisons — we started with the first item, compared it to the second adn foudn which was smaller, then compared that to the third and so on. So we've discovered certain comparison relations; why don't we at least record those for future use? SO when we compare x_1 and x_2, we draw an edge recording the outcome of the comparison ($x_1 \to x_2$ means $x_1 > x_2$, so x_2 i the current candidate). Then we compare this to the next guy, and maybe it wins again, so we also draw an edge $x_3 \to x_2$. This may hapen several times (x_2 is a winner beating tow things), and then it may lose ot something else. Now we have a new leader who will win a few contests and then lose to something else. This will happen repeatedly.

So when we find dthe minimum we will have implicitly constructed this set of outcomes from teh tournament we ran. SO we might as well make it explicit and save it for next time, so we don't forget the work we did.

The thing we construct is a tree (everyone has one parent, the node it lost to). It's actually a special kind of tree, a *caterpillar* (graph theorists love giving poetic names to the shapes they come up wth, like there are actually bicycles). This has a spine with heads hanging off different elements; and the minimum is at the root.



So we found teh minimum but also all this stuff.

However we have been told ot delete the minimum — after mkaning this caterpillar, the first thing we do is cut off its head.

But we're much better off than if we didn't remember anything — because the next minimum is one of hte children of the root. Our plan is to use the work we did — on the next delete-min, the min will be a child of the old root.

So by being smarter about the work we do, we can reduce future work. How much reduction of work can we achieve this way? We do have to worry about inserts interleaved with delete-mins, but we won't do this right now. (If we have an insert at this point, we can leave it as another child of this root.)

Optimistically it could be that everything is in a neat order and the cateripllar is a straight line; the worst case is that the caterpillar is a spider (the formal name is a star). If the input arrives sorted, we do $O(1)$ work. If it's a star (the first thing we looked at was the minimum and it keeps winning, so we get a star), then we gained nothign.

Generally speaking, what determines the amount of next work was the number of children of the head — the degree of the root. That points at where we'll need to be smart next — if the work si determined by the number of hcildren of the orot, we have some flexibility in comparison order, so can we do comparisons that ensure we end up with not too many children?

We'll take a break at 1 hour, and then discuss how to make sure you don't have too many children.

(During break David got many questions; this is not good, you should ask them during class. THe COpernican principle says it'd be relaly weird if only Earth had conditions leading to life; similarly if we have questions, likely many others do too, so you should ask for the benefit of other people in the class. Then you can use break to talk to your friends.)

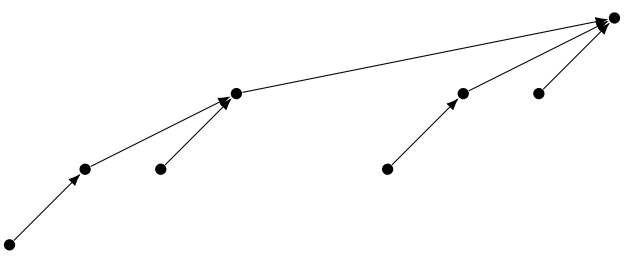
So now we are motivated to think about how to not have too many children of a node getting deleted. What can we vary to control the number of children? WEll, we talked about how to do comparisons by keeping a current minimum and comparing it to other things; but you don't have to do your comparisons in this way.

The problem when we cause a star is a minimum element plays in too many competitions and keeps winning. What we might do instead to ensure no element plays in too many competitions, so that no elemnent can possibly win too many? You could pick random pairs, but the randomoized algorithms class is next year. You coudl have multiple groups compete against each other, as in sports when we want to balance competitions — we break things into groups and have different groups compete against each other. For exampe we can think about a canonical tournament where you have n players, pair them and have them compete, pair up those and have a next round hwere now you get winners of 4, then 8, and os on, until the semifinals nad then finals.

WE are going to use that kind of tournamnet, where we have pairs compete and then the winners of those pairwise competitions compete, and those winners ocmpete with each other, and so forth. WHat kidn of competition tree does this create?

The tree we constructed is what wel'l call a *heap-ordered tree* for obvious reasons, it's a tree where each element is smaller than its children.

If we run a tournament like this, we get a binary tree:



So we've got a bunch of leaves (the losers of the first round). The number of nodes is — we did individual nodes, then pairs, then groups of 4, then 8, and finally a total of 16 nodes — so these tree shapes are for power-of-2 number of nodes.

What's interesting bout this shpae? WE've got degree-0 nodes with no descendants; how many descendants a degree-1 node has is 1 plus itself which is 2, a degree-2 node has 4, in general a degree-d node has 2^d descendants (including itself). There's something else fun about the shape of this tree. If we look at a pair, in a set of four nodes you have levels with 1, 2, 1; with a gorup of 8 the number is 1, 3, 3, 1, and then 1, 4, 6, 4, 1, and os on. So these are *binomial trees*. And what we've just demonstrated is a n-node binomial tree has max-degree $\log n$. (All logs in this calss are base-2 unless otherwise specified.)

These are hte fuhndamental component of *binomial heaps*, which improve on standard heaps but not enough (so we won't talk about them).

But inserts might also happen in the middle of deletes. SO instead of having just one hea-pordered tree, we'll have many. WE go back to being lazy — if someone gives us another insert, what do we do, we just drop it on the side. Now I have two heap-ordered trees. I get some more inserts and drop them in as well, making more heap-ordered trees. Eventually I get requested delete min and I need to do some more work.

Now I have many things and I need to compare their roots. So on an insert, I add a heap-ordered tree. On a delete-min, I run a tournament of some sort to compare roots — I consolidate the heap-ordered trees by competing their roots. Then I end up with one heap-ordered tree, at what point the minimum is the root, I delete it and end up with more heap-ordered trees.

But we need to be careful how we do this competition among heap-ordered trees. What oculd go wrong if we just decide we'll collect these nodes and then run a competition? The bad outcome is if I start with a big heap-ordered tree and compare to lots of other nodes, those could all lose and I could end up recreating a star. So I don't want to do that; I need to exercise the same sort of discipline as I did when building a perfect binomial tree.

So we want to be careful not to create stars or anythign like them. The way we'll do this is using *union by rank*, a fancy name developed by Tarjan (who also developed the union-find data structure) .The idea is that we are going to record each node's degree, and we are only going to compete nodes of teh same degree. We claim this is going to give the result that we want.

By induction, if we only link same-degree nodes, then a degree-d node has 2^d descendants, which means the maximum degree is $\log n$. In binomial heaps follow this pattern very aggressively — they keep binomial trees around and only apir those up, when you delete min you always have a collection fo binomianl heaps and this invariant always holds.

Fibonacci heaps are sloppier because they're lazier, which makes them better. WE'll now formalize how we do this ocnsolidation for arbitrary heap-ordered trees.

Our inserts ansd deletes — we're going to maintain some *buckets* where we bucket HOTs (heap-ordered trees) by their root degre. THen to do consolidation, we start at degree 0 (or the smallest degree where we have anything in the bucket), and we compete pairs and promote htem to the next bucket. This is so we don't have to go hunting for a tree of the same degree to run a competition; wej ust keep them bucketed at athe beginning.

WE do this until there are less than two trees left in the bucket (at most one remaining tree).

Then we go on to the next bucket (the next larger degree). And we keep doing that through all the buckets.

(At a certain point there will be 0 or 1 trees in the bucket, and that's when we move on to the next.)

NOw how do we implement delete-min? All we do is remove the min, move the children to the proper buckets, and then we consolidate to find the next min. We're not being quoite totally lazy — we could wait for consolidation until we get to hte next delete-min — but if we consolidate now, tehn we can support find-min. (That's an unimportant detail.)

Now we cobine with lazy insert — to support insert, we just add a one-item HOT to bucket 0.

One other detail — the implementation of the HOT, we need to take a node, get to its children, be able to cut off its children from that node, and so on. Being a lazy theoretician, we just put lots of doubly linked lists as the implementaiton; this has the nice property — we link every node to its left child, and make a doubly linked list between the siblings. The advantage of this representation is every node has excatly two sibling pointers, one parent, and one child, so they all have the same size (heap-ordered trees can have arbitrary degrees, so how to represent that).

HOw to analyze time fo rthe operations? Because we're being lazy, the time for any particular operation could be huge (if we have lots of nodes in bucket 0 after n inserts, then the fist delete-min makes us consolidate everything). But we'll do an amortized analysis hwere the only way for us to be forced to do lots of work is if the adversary did a lot of work. WE use a potential function representing how complicated the data structure is (how mcuh work we've put off for future us).

So we define φ as the number of heap-ordered trees. This is our potential function.

WE define

$$\text{amortized cost}_i = \text{real cost}_i + \text{change in potential} = \text{real}_i + \varphi_i - \varphi_{i-1}.$$

If potential increased, then amortized cost is larger than real; if it decreased, it's smaller. The benefit of this is

$$\sum a_i = \sum r_i + \varphi_n - \varphi_0.$$

And that lets us say $\sum a_i \geq \sum r_i$ as long as $\varphi_n \geq \varphi_0$. One way to achieve that is to have $\varphi_0 = 0$ nad $\varphi_n \geq 0$. This is typical of potentail functions, and it's true of ours — we start with nothing, and we always have something in the data structure. So just by defining a potential function, we get something that upper bounds the total cost of all the operations we perform.

What are the costs of the two operations we've defined? The amortized cost of an insertion operation is 2 — the real cost is 1 (we do a ocnstant amount of work), and the change in potential is also 1.

What about deletion operations? Well, the real cost of delete-min is — well, we're starting with r roots. WE're then deleting the min, which adds c children to what's there; and then we do $r + c$ scan-work to consolidate. (If we have a certain number of nodes, every comparison decreases the number of roots by 1.)

And then we end with at most $\log n$ roots, because there's only one per bucket. So to tal kabout change in potential, we started wtih r roots and ended with $\log n$; this means the change inpotential is $r - \log n$ (thsi is a decrease). So if we sum the $r + c$ real work with subtracting the $r - \log n$ change inpotential, then the amortized cost is

$$r + c - (r - \log n).$$

That means the amortized cost of delete-min is $c + \log n$.

But hwat can we say about c? That's at most $\log n$. So the amortized cost of insertion is 1, and of delete-min is $\log n$. So just with this one little lazy trick, and a potential function analysis, we've beaten binary heaps (which od lots of wasted work on insert even though you might never get delete-min; this is smarter and saves the work for delete-min, and this doens't cost anything but a constant factor).

So that's a start, but not enough for what we wanted — because we haven't yet talked about decrease-key,whihc is where all the work is in Prim. So we'll next take the idea of taking this idea of a collection of HOTs and consolidating only when you need to, and how you'd generalize that to support decrease-key.

§2 September 6, 2024

§2.1 Review

Last time we took a leisurely path to motivating Fibonacci heaps; let's review where we stand. The story here is about laziness and procrastination, which David recommends as a general approach to life (but not

in this class). We developed a heap as a collection of *heap-ordered trees* (HOTs), plus a pointer at the minimum (that we keep around so that we can answer `find-min` queries in $O(1)$).

On `insert`, we're extremely lazy; we just add a tree, and possibly update the min-pointer if necessary. This laziness is one of the reasons we immediately need to think about *collections* of HOTs — each insert produces another one.

Things got interesting on `delete-min` — when we perform `delete-min`, we need to find the *next* minimum, and as we do htis, we *consolidate* the HOTs into a (likely smaller) number of HOTs — because in the consolidation, the work we're doing (the comparison of roots) is something we'll have to do *anyways* to find the next min. So this is a pretty natural, generic approach — wait until a `delete-min`, and when you get one, you finish up the tournament among the roots of the HOTs you have to find the current min, and also to reduce the complexity of the data structure.

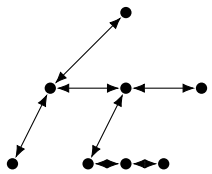
So the only question that had to be addressed was, how do we consolidate for good performance? Aside from housekeeping, pretty much any consolidation algorithm is going to be linear in the number of roots — every time you do a comparison, you're going to combine two HOTs into one, so you'll have to do roughly the same number of comparisons as HOTs. So there aren't performance opportunities in how you do the consolidation; but there are in *which* comparisons you do.

For that, we apply the heuristic of *union-by-rank* (which is one of Tarjan's big ideas; it shows up in the union-find data structure as well), an approach to keeping things balanced — we only compare heap-ordered trees of the same root degree. When we consolidate them, we take two HOTs of degree k, and one gets a new child, so it becomes a degree $k+1$ HOT. So we get by induction that a degree-k HOT has 2^k nodes. And therefore, the maximum degree is $\log n$ (if we have a n-node heap).

This was useful because the degree of a node tells you the number of trees that you add when you delete the root of the minimum HOT.

> **Remark 2.1.** Couldn't you get this property even by doing a tournament? Yes, but the point is that union-by-rank allows us to control *when* we do the comparisons — imagine that we have a tournament that's been running for a while, but then some nodes show up late. You could compete them against the old winner, but that winner is tired; you compete them against each other first.

Then we talked a bit about how we implement the data structure, where each node gets a parent pointer and a first-child pointer and some sibling pointers, which allows you to represent the heap-ordered tree using at most 4 pointers per node. (It's in general nice to not have to change the size of your basic nodes.)



This lets us move between parents and children, as well as to remove a root and immediately collect its child nodes.

We won't go through how we do consolidation by rank (we have buckets, one for each degree, and when we run a comparison within a bucket, we promote the result to the next bucket).

At this point, we did the analysis using a potential function φ. The idea is that a potential function is a function on the data structure — it can be anything you like — but we want one that works for us. Then we define the *amortized time* as

$$\text{amortized}_i = \text{real}_i + \varphi_i - \varphi_{i-1}.$$

Then we have

$$\sum a_i = \sum r_i + \varphi_n - \varphi_0.$$

This means that $\sum a_i$ (the amortized cost) is at least the real cost, so long as $\varphi_n \geq \varphi_0$. For example, this holds if $\varphi_n \geq 0$ and $\varphi_0 = 0$, which is very common — empty data structures generally have 0 potential.

And this means we get an upper bound — to analyze the real cost of a series of operations, we can instead compute the amortized cost. (All the charging schemes you've seen for red-black trees and similar things can be turned into a potential function argument; potential functions are very general.)

We defined φ as the number of HOTs in our data structure; this fits with the goal of having φ represent the amount of 'deferred work' that you may have to do in the future. If $\varphi = 1$, then this means we have a HOT — we have no work. If φ is large, we've got lots of HOTs, and when `delete-min` happens, it will take lots of work for us to figure out the next min.

Then what's the amortized cost of `insert`? The real cost is $O(1)$ (we just add a heap-ordered tree of one node). The change in potential is $+1$, since we added one heap-ordered tree. That means the amortized cost is $O(1) + O(1) = O(1)$.

Now when we consider `delete-min`, suppose we have r roots (or heap-ordered trees) and the min being deleted has c children. The real cost is $r + c$ — we have r roots and then the c children of this old minimum all become their own roots, so we have $r + c$ roots, and then we spend $r + c$ time running consolidation. Meanwhile, what can we say about the change in potential? The initial potential was r (there's r roots), and the final figure is at most $\log n$ (since there's at most $\log n$ buckets, and each bucket ends up with at most one heap-ordered tree left). When we put these together, we get

$$(r + c) + (\log n) - r = c + \log n.$$

And we also have $c \leq \log n$ (because we don't allow any node to have too many children), so this is $O(\log n)$.

So that's where we left off — we have a data structure that supports $O(1)$-time insert and $O(\log n)$-time `delete-min`, so it's already better than a binary heap. We can also support `merge`, where you have two completely unrelated priority queues and want to smash them together into one.

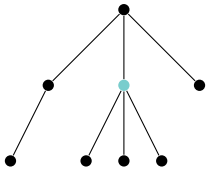
(Note that for most of the time, we keep a linked list of heap-ordered trees; we only create buckets when consolidating.)

This takes $O(1)$ work — we're just concatenating two linked lists. And the change in potential is 0 (if you think about the potential of the two heaps separately). So `merge` is another operation you can perform in $O(1)$ time; this is again a wonderful improvement over binary trees.

> **Remark 2.2.** The generic state of the data structure is a linked-list of heap-ordered trees, with a pointer to the minimum. (For `delete-min` we walk over the linked list, putting everything into the appropriate buckets, and then we consolidate.)
>
> (Walking over this list is linear, but most of that is paid for by the destruction of the forest — we are destroying a forest to get the energy to pay for the work we're doing.)

§2.2 Decrease-key

We need one more operation to use these heaps for Djikstra and Prim and so on, which is `decrease-key`. We have some heap-ordered tree, in some sort of shape; and someone comes along and says, you know what, I want to decrease the key of this node sitting somewhere deep in our tree. (We can assume that we have a pointer to the node in the data structure — if you're using the heap for something, we have a heap of elements (e.g., nodes from a graph), so our heap will be a collection of pointers from the heap to vertices of the graph. It's easy to make that doubly linked, so given a graph element, we can easily get to the heap element.)



We can easily just decrease the key, but that'll violate the heap-ordering property — this new key could be smaller than all sorts of things. So what should we do if we want to restore the state of our data structure?

If you think from binary heaps you might want to bubble it up, but that takes a lot of work, and we are lazy. So we are just going to cut it off and promote it to being its own heap-ordered tree. We're already designing things so we have a collection of roots; we just have one more now.

This is awesome — it's $O(1)$ work, which is fantastic. If you can do this, you can do all sorts of amazing things.

But in fact, the runtime is *not* $O(1)$. It is true it only takes $O(1)$ real work to do this cut-and-promote. But what goes wrong if we do this? The problem is if we're going to start reaching into the bottom of the tree and pulling out subtrees, we lose the invariant that a degree-k heap-ordered tree has 2^k nodes.

So what are we going to do about this? Back in the day, there was a movie called 'Saving Private Ryan.' This is based on a true policy that the army has, that once a family has lost too many children in war, they don't send the other ones to war (you don't want any one family to lose all its children). That's the principle we're going to apply here as well — after a node loses a child or some children, it becomes more important to preserve the other ones. So we're going to try to change our approach to `decrease-key` to give more priority to nodes that have already lost children.

> **Remark 2.3.** We want to maintain a *lower* bound on the number of descendants of a high-degree node, because if it can have few descendants, then you can have really high degree.

The solution is actually pretty simple. Our rule is going to be that when a node loses a *second* child, we are going to cut *it* from its parent and make it a root.

We can implement this straightforwardly by adding a mark-bit to each node (which gets set to 1 when a node loses its first child). We set the mark-bit when you lose a child, and clear it when you lose your second child and cut yourself and promote to root.

This can have some consequences — that might cause *that* node's parent to lose its second child as well (and then maybe that's the second child of the grandparent and that guy needs to be cut as well). And this may go up all the way to the root. So you have the potential for *cascading cuts*, where you walk up an entire path to the root of your tree, cutting all those off.

So on one hand, this seems like lots of unnecessary work. But the point is that it's actually necessary to maintaining the densities of the trees (to keep them from getting too sparse).

§2.3 Analysis

That's actually the only change that needs to be made; all the fun comes from analyzing what this does to the data structure.

The first thing we want to argue is that despite the huge amount of work we seem to be doing, cascading cuts are actually free — they don't cost any real work. So this means we want to make sure our potential function pays for any of the work we're doing in cascading cuts.

The second thing we want to prove is that thanks to these cascading cuts, tree size remains exponential in the degree.

As long as these two things are true, everything else from our previous analysis is unchangd. We still have $O(1)$ insert, $O(\log n)$ delete-min, and $O(1)$ merge; and since cascading cuts are free (the only thing you may have to do is update the min-pointer and change the key value), it'll turn out that the other operations are unchanged and we get an $O(1)$ `decrease-key`. And that's the real killer for Fibonacci heaps — you can decrease keys in $O(1)$, and this makes a huge difference to Djikstra and Prim.

§2.3.1 Cascading cuts

We have two things to show; let's start with the argument that cascading cuts are free. We want to make a potential function that measures the number of deferred cutting work; so we can take φ to be the number of marked bits. Then when we do a `decrease-key`, we start at the bottom, and we've got some constant amount of real work; we take the first node and cut it. But now we look at the parent; if the parent is marked, we need to cut it. But when we cut it, we clear the parent's marked bit, which pays for cutting the parent. Similarly, we may have to cut for the grandparent, but only if it has a marked bit, and then that pays for it. So each node that is cascade-cut (other than the first one) is already storing the potential that is going to pay for cutting that node — it's marked, and it gets its marked bit cleared, which means that it's already paid for.

That's a bit too simple; there's a problem, which is that we just said our potential function was the number of roots, and we can't just change to a different potential function in the middle (that's like throwing energy into the system). So we can't just use this potential function; we have to add it on in some way.

But this isn't hard. Note that we can't just add our two functions together, because we're clearing marked bits but also making new roots. But to fix this, we're going to double the marked-bits potential so that it dominates — we define

$$\varphi = 2\#\text{marked bits} + \#\text{roots}.$$

We can use this potential function all the way through our analysis (for all the other operations, they don't affect the number of marks at all, so the roots piece of the potential pays what it's supposed to). Each time we do a cascading cut, we get two units of potential; one cancels out the work of the cascading cut, and the other cancels out the increase in the root potential. So each cascade gives two units of potential; one pays for the cut-work, and the other cancels the change in the root potential. This means cascading cuts really are free.

§2.3.2 The exponential-in-degree claim

We now get to the climax (answering an earlier question of where the name comes from).

Let's take a node x, and consider its *current* children (we forget about the ones that were lost), in the order that they were added; let's use 1-indexing (so you have the first addition, the second, the third, and so on).

> **Claim 2.4 —** The ith added child (of the ones still remaining) has degree at least $i - 2$.

This may seem mysterious, so let's prove it (that'll make it less mysterious).

Proof. Let's call y the ith added child. What can we say about the state of things when y got added? This must have been added during the consolidation phase (that's the only time you add children), and you only

pair up nodes if they started with the same degree (that's the reason you compared them and made one a child). And so this means before that, y had the same number of children as x did (when it was added). And y is the ith added child — all the $i-1$ earlier children were already there. So x had at least $i-1$ children at that time (it may have had more that got removed later, but it had at least these). And that tells us that y had also at least $i-1$ children at that time.

What does that tell us about now? We know y can only have lost one child, so it still has at least $i-2$ children now. □

Remark 2.5. If a child gets added to the same parent multiple times (by getting cut off and reconsolidated), we consider the *last* time it got added.

Now that we've got this lemma, how do we use it? Let s_k be the minimum possible number of descendants of a degree-k node (including itself). Using this lemma, we can write a recurrence! We have $s_0 = 1$ (any node has itself as a descendant), and $s_1 = 2$ (this node has degree 1, so it has a child, which means there's at least two nodes).

What is s_2? This means I have a node, and it has two descendants. One was added first, and that has at least $1-2=-1$ children; this is not particularly useful. The second node has degree at least $2-2=0$, which is also not very helpful. So this tells us $s_2 = 3$.

Now degree 3 is where things get interesting, because the third node has degree at least 1 — it has to have a child. So this takes us to $s_3 = 5$.

More generally, you can say that any node of degree k has — you can count itself and its first child, and then if you go on with the other children, you get

$$s_k \geq 2 + \sum_{i=0}^{k-2} s_i.$$

To solve this, we can set them to be equal. And then we can simplify by subtracting

$$s_k - s_{k-1} = s_{k-2}.$$

(We have the same thing on both right-hand sides, except for where the sum ends.) In other words,

$$s_k = s_{k-1} + s_{k-2}.$$

And that's why they're called Fibonacci trees.

But more importantly, we know that $s_k \geq \varphi^k$, where $\varphi \approx 1.618\ldots$ is the golden ratio (not the potential function). In particular, s_k is exponential in k — which means a degree-k node has at least φ^k descendants, and so the maximum degree of a k-node tree is $\log_\varphi k = O(\log k)$.

So with this little 'Saving Private Ryan' adjustment, we have preserved this $O(\log n)$ max-degree condition; and that's the only thing we needed in our proof that `delete-min` cost $O(\log n)$ time. So we're done!

And that's the story of Fibonacci heaps.

There's always a coda; this coda is kind of long, so we'll save it for after the break.

§2.4 A coda

One question David finds very interesting (though some theoreticians don't) is whether this data structure is practical — would you ever want to use a Fibonacci heap? The constants are not that bad — we haven't

done anything crazy (our constants have some factors of 2, but not 10^7). So Fibonacci heaps do in fact reduce the *comparisons* that are needed, even for moderately-sized problems.

However, you still have to worry about the bookkeeping overhead of moving pointers around and joining linked lists and stuff; that might cost you in practice. Years ago when David used to do research in algorithm implementation, they ran experiments on a basic graph problem (Min-Cut) with a number of data structures, including Fibonacci heaps. Their experiments were on computers much dumber than our smartphones are now; and there Fibonacci heaps won in some cases, even on the tiny things we could fit on computers 20 years ago.

But in general, there are several reasons for you to be skeptical about the utility of Fibonacci heaps compared to regular heaps. One of the main points is that a regular heap (a stanard binary heap) is in an *array* — it's what's known as an *implicit* heap (you don't have to move pointers around, you just have elements in the array and you move the elements around). In contrast, Fibonacci heaps do lots of pointer manipulations. And pointer manipulations are bad in modern computers, because you get bad locality of reference. We'll come back to this when we study online algorithms; basically they have caches, and when you fetch data it sticks around for a while and the nearby data does too. In the binary heap, the top is in a few cache lines, so as long as you're just accessing that, binary heaps are very fast.

Pettie has developed an optimal *implicit* heap which matches the time bounds of Fibonacci heaps, but it's complicated enough that its constants are probably going to kill you (even though it has better locality of reference).

§2.5 Back to MSTs

We'll now have a second coda, where we go back to the motivating example of minimum spanning trees. If we go back to Prim's algorithm, there's n inserts, n deletes, and m `decrease-key`. In the first lecture we tried to do balancing games between them, but here we don't need to — `decrease-key` is already $O(1)$. So we immediately get MSTs in $O(m + n \log n)$ (as opposed to $m \log n$ from all the naive algorithms). This means this algorithm is *linear time* for all but the sparsest graphs.

This is great, but theoreticians always want more — what can we do to improve on the runtime for sparser graphs? A theoretician looks at this beautiful runtime and asks, why is it so slow? What's the bottleneck? The bottleneck term is the $n \log n$, coming from the `delete-min`. And it's slow because there are n items in the heap.

So this suggests how we might overcome the slowness — if we can keep the heap *small*, then we don't need to pay that $\log n$ cost. This is an algorithm developed by the same paper of Fredman and Tarjan (in the same Fibonacci heaps paper).

We're going to choose a max heap size k. And the idea is we're going to run Prim's algorithm until our heap has k elements, or alternatively until the tree that we're growing has k neighbors in the graph. (Naive implementations of Djikstra and Prim initially put all vertices into the heap with ∞, but you don't need to put anything into the tree until you reach it — when you add a vertex you look at its neighbors and see if they're in the heap, and if they're not then you put them in. Then the size of the heap is just the boundary of the thing you're growing.)

So then we'll be running Prim, but paying $\log k$ instead of $\log n$. Eventually, though, we're going to end up with k elements in our tree. What do we do now? The idea is to just go somewhere else! If you've destroyed the environment in your local neighborhood, you can just move. So you restart at a different node and do the same thing.

This is kind of Kruskaly — there you're taking edges from wherever they come from (the rules for MSTs can be applied locally). We grow that tree until it gets k nodes, *or* it attaches to a previously abandoned tree. (We're growing by adding edges out, and one of those edges might connect us to a tree we grew previously.

That's fine, because if we've attached to a tree with more than k nodes, then the combination certainly has more than k.)

We repeat this over the entire graph — for every vertex we check it, and if it's not yet in the tree then we keep growing it until we get a large component, and so on. So we're scanning all the vertices and trying to grow these trees as far as we can. We continue this until all the vertices are in these trees, where each tree has k incident edges.

What should we do now? We've basically gone over our entire graph and built up these little trees that each have k incident edges on them. Now we're going to be Kruskally — we just draw a circle around each of these trees, and, well, we still have edges between the trees, but now we can think of them as edges between the circles. And to finish the MST, you just need to find a MST in this contracted graph!

So we do what's called a *contraction* step, which means we merge each tree into one vertex with all incident edges (if this creates multiple parallel edges, you just take the smallest of each parallel copy). And then we recurse — but we recurse with a new value of k (since after this contraction, every vertex already has k incident edges).

Let's now talk about the schedule of evolution of these k-values. Formally, we have a phase that starts with some number t of vertices (which we got by contracting the forest from the previous phase). We pick a region size k, and then we grow trees until all the trees have k incident edges. And then we contract, and repeat.

Let's think about the runtime of these different stages. If we're going to grow trees until all trees have k incident edges, how many data structure operations of each type do we perform? We have t vertices, so we do t `insert`s and `delete-min`s, and m `decrease-key`s (we don't know much about the change in the number of edges, so we still use the bound m). But the heap sizes are always at most k, so `delete-min` only takes $\log k$ time each. So our runtime is

$$m + t \log k.$$

(You can do contraction in $O(m)$ by e.g. BFS.)

> **Remark 2.6.** The number of trees doesn't actually matter — there are t vertices and each will be inserted and deleted once, and the m for `decrease-key` doesn't depend on how many trees there are. (The t sums up over all our trees.)

And we can observe that since we're spending m, we might as well have $t \log k = m$, because that doesn't change our asympttic runtime at all. So if we invert that function, we should set $k = 2^{m/t}$. And if we do that, then the runtime of a phase is $O(m)$.

Now we just have to ask, how many phases are there? At the end of this phase, each contracted vertex (i.e., each vertex for the next phase) has at least k edge endpoints attached to it (the reason we stopped is that there's k distinct edges sticking out). And there are only m endpoints in total (really $2m$, but you can argue it down to m). That means the number of vertices in the next round — since each has to have k endpoints — is at most $\frac{m}{k}$.

And if the next t' is $t' = m/k$, then $k' = 2^{m/t'} = 2^k$. So in each phase, we don't add to k or double k; every phase *exponentiates* k. So the number of phases is equal to the *height* of a tower of 2's that equals n — how many times you have to exponentiate 2 to reach n (or actually, you can start at m/n because then one more phase finishes up). Formally, this is the function $\beta(m, n)$ — the smallest number i such that

$$\underbrace{\log \log \log \cdots \log}_{i} \frac{m}{n} \leq 2.$$

And this is at most $\log^*(n)$, which is a very small number. (You can imagine repeated exponentiation grows things very quickly.)

This seems like a very small number, but theoreticians are never satisfied. A few years later, Tarjan and some other people improved this to $\log \beta(m, n)$. This is a lot better than putting one log on the $\log^*$ — you do this thing that makes a number tiny, and then take a log of that number. The idea was to keep edges in a bundle, and only look at the smallest edge in a bundle until you used it up.

Bernard Chazelle (whose son Damian Chazelle directed Whiplash, which is an awesome movie) actually improved this to $m\alpha(n) \log \alpha(n)$, where $\alpha(n)$ is the famous inverse Ackermann function, which is so much smaller that it makes this small function $\log^*$ look huge ($\log^*$ is 7 for the number of atoms in the universe, and α is 4).

But we're still not done; what is the optimal runtime? Pettie and Ramachandran developed an *optimal* algorithm, and proved that it's optimal. Unfortunately, we don't know what the runtime is. How can you prove an algorithm is optimal without actually knowing its runtime? The point is you posit the existence of the optimal algorithm, use brute-force search to write down all the possible algorithms on tiny graphs, and then — this is so close to linear that if you just make the graph a bit smaller, then this is good.

But we do know thanks to Komlos that you can find the MST using only $O(m)$ comparisons; this is much better than sorting. Unfortunately, Komlos's algorithm, in order to figure out what those comparisons are, uses n^3 time.

Finally, there is a *randomized* algorithm (David teaches that next year) which is *linear* in expected time. So we've *almost* killed this problem.

Everything we've talked about here fails for Djikstra; you can't get close-to-linear Djikstra.

§3 September 9, 2024

Today we'll talk about two amazing data structures — the first is amazing in what it does, and the second in how it does it.

§3.1 Persistent data structures

The problem persistent data structures address is: we've got lots of data structures, which support lots of operations for updating them. But in a typical data structure, once you update it, you've lost information about its past. And often we'd like to know about the past, or to be able to ask 'what was the smallest element of this set a long time ago'? You might even want to unroll your data structure and revert it back to a previous point.

> **Question 3.1.** How can we take data structures and make them *persistent* (so it's possible to access their pasts)?

This was introduced by Sarnak and Tarjan in 1986. Their paper was called *Planar point locations using persistent trees* (we'll see this as an interesting application).

There are a couple of ways to approach persistent. Usual data structures are *ephemeral* — you can only query and update in the present. You might want what we're going to call *partial persistence*, where you only update the present, but you can query the past. But to be mindbending, Tarjan also thought about techniques for *full persistence*, which allows you to 'do time travel' and both query *and* update the past. Updating the past, as we know, is very dangerous and has all sorts of unintended consequences, creating puzzles about what's the real timeline, and you get branching structures. Full persistence is hard to think about, and we're not going to do so; but they have techniques that achieve it in some cases.

We'll see general techniques that apply to a broad range of data structures — specifically, data structures that fit into the *pointer machine model*. This means the data structure is made of nodes with a constant

number of fields that hold values, and also pointers to other nodes. For example, a binary search tree is a pointer-based data structure; a heap the way we usually implement it in an array is not, since arrays are these arbitrary-sized things you can index into. But you can implement a binary heap or a Fibonacci heap using pointers, so they also fit into this model.

We'll look at techniques for making them persistent. When you navigate these structures, you do so by way of following the pointers; so any exposed operation is implemented through some sequence of primitive operations of reading and writing fields in the nodes. So in order to make this persistent, we're going to figure out how to make these atomic actions — reads and writes of fields — persistent. If each of these modifications is a persistent modification, then we'll be able to get persistence of the data structure as a whole (if we have the old values of all the pointers, then we can also look up the old values in the data structure).

The work of Sarnak and Tarjan looks at arbitrary pointer-based data structures, but the notation gets messy, so we're going to focus on trees (which gets all the basic ideas across without getting us lost in the notation).

§3.1.1 A first attempt — the fat nodes method

Once we've set it up this way in terms of basic collections of fields in each node, if we want to think about keeping the prior state of the data structure, the obvious thing is to make each of these fields independently persistent. We can do this in an obvious way — just replace each field with a log of timestamped field changes. (Every time we update the field, we simply record the new value in a log.) If that's what we're going to do, then each of our fields is now a giant log of changes to that field.

Then how do we navigate the data structure? A node has two child pointers, but now each is a log of all the values that child-pointer ever had. So if we want to do a lookup (where we're given the time at which we want to perform the lookup), how would we do it? We've got an ordered list of timestamps (e.g., as a binary search tree or array), and we can use binary search to look up the value of the pointer at the right time; then we follow that pointer, and that takes us to another node. So each step through a pointer in the data structure is going to involve a binary search on the values of the field we're looking at.

What about a modification? That's easier — we just add one more entry to the log for the field we're editing.

What's the effect of this on the data structure? Well, now we have two things to think about — we want to think about the time to perform data structure operations, but also the space it takes.

What's the effect on space? Each time we make an update, we have to add a log entry; that's a constant amount of additional space (a value and timestamp). So each atomic change (`write`) costs $O(1)$ space. What about time? Each atomic `read` now takes $O(\log t)$ time, where t is the number of updates. (You can think about this in terms of number of updates or granularity of your clock, but it's some measure of the number of changes.)

These are the effects on atomic operations (reads and writes). But what does this mean for a binary search tree, where we're not just doing a single operation but navigating through a whole series of fields? If we think about a balanced binary search tree, what does this do to the runtime for a lookup? Now the time is $O((\log n) \cdot (\log t))$. In data structure land, logs are kind of the bread and butter — we expect things to be logarithmic — so this is not great (it's kind of like going from n to n^2 in an algorithm).

And what about space? If we perform an `insert`, this can change $\log n$ different nodes (depending on the implementation of the balanced binary search tree); and $O(\log n)$ atomic changes mean we're using an extra $O(\log n)$ space per update. This is also kind of worrisome — we did one update, and got a multiplicative factor in update cost.

So these are both things we'll try to improve on. (Our specific goal is to make a persistent binary search tree, but the solution will generalize.)

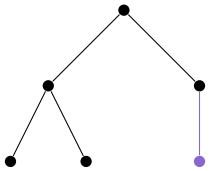
The $\log n$ space here isn't actually the fault of the persistent data structure; it's the fault of the underlying one — per primitive operation we're using $O(1)$ space, and you can't beat that. But what *does* seem to be avoidable is the multiplicative $O(\log t)$ in the runtime, and we'd like to avoid that.

We call this the *fat nodes* method because we're making each node very large, to have room for each element of the log.

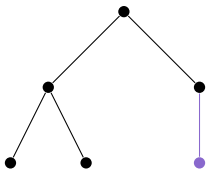
§3.1.2 A second attempt — path copying

Now we'll talk about a different method, which is maybe less obvious. In a typical structure, you have some root and follow a series of pointers, and you can only get to a node by following pointers. If other parts of the data structure change, this doesn't matter if nothing along your own path has changed.

Imagine we've got a little binary search tree at time t_1, and suppose we now want to add a node on the right.



What part of the data structure is affected by the change? It's only the part that has a path to the new node. So what we can do is copy just the path to the new node — we make a new copy of the part of the data structure that has changed, but recycle the old parts of the data structure that have not changed.



What are the benefits and downsides? Lookup is significantly improved — you no longer have the $O(\log t)$ multiplicative factor. We're going to still have t roots, but once we get to the proper root, our nodes aren't fat — they're the original notes. And then once we get to the right root, we can just use regular data structure operations to find the right node. So the runtime for lookup is now $O(\log n + \log t)$ (where $\log t$ corresponds to finding the right root, and $\log n$ to do the regular lookup). This is a dramatic improvement (e.g., going from n^2 to $2n$).

What cost does this come at? The cost is memory — every time we do an update, we're copying an entire path. So this could use possibly a huge amount of space per update. How much depends on the shape of your tree, but if you're adding one node, even though technically one pointer may be changing, we're nevertheless copying the entire path.

So fat nodes are great for space but bad for time; path copying is great for time but bad for space. We're going to try to get the best of both worlds.

§3.1.3 The solution — lazy path copying

The problem is that we're not lazy enough — we've got a new value at the leaf, and we're already making a copy of lots of stuff. We'll use a technique that uses more procrastination as to the allocation of space. (This is the thing that Sarnak and Tarjan came up with.)

Question 3.2. How can we be lazy about path copying?

In our fat nodes method, we said fat nodes are bad, but we know there's a healthy way in the middle — slightly stocky nodes are not such a big deal. So we're going to let nodes get *slightly* fat before we start copying. We mean *really* slightly — each node will have *one* extra timestamped field (not twice as many, not one per field of the node, just one extra space).

The first time any field of the node is changed, we will use the extra field to record the identity of the changed field, the new value, and the time at which it happened — so there's a slight increase in the space of the node.

The second time, we create a new node as a copy from the old. But we will incorporate the new field — we make this copy, but include the value from the overflow field in the field that it replaced. So we incorporate the value from the extra field. We then perform the new atomic change on this copy. Note that we now have a new empty extra field in this copy.

But now we've made a copy; if we copied a leaf, now we have to update the parent to point to the copy. How do we update the parent to point to the copy? Well, it's the same problem, and we do it in the same way, recursively. So we think of this as a time-stamped update to the field pointing at the copied node; and we do that in the same way (if there's a free field in the parent then we use it; if there isn't, then we make a copy of the parent, and then update the grandparent). This may remind you of marking and cascading in Fibonacci heaps — one update down at the bottom might cause a propagation of updates all the way up to the root of the data structure.

Obviously that could be very expensive in both time and space — for example, a linked list has length n, so if you update the tail of a linked list, you could end up doing n work to copy the entire linked list (to reflect one change at a tail). This is going to be very bad from a cost-per-operation perspective, until we think about *amortization* — we're going to argue that even though you might do a ton of work on a particular operation, the average cost per operation is small.

If we took this approach and didn't worry about space, what would the runtime effect be? How does this change the runtime of a lookup? It's the same idea as path copying — we have to find the right root (delayed path copying does result in eventually making copies of the root, so we need some structure); but then as you descend from the root, you have $O(1)$ multiplicative overhead (in every node, you do have to check the extra field to see if it overrides the field you're planning to look at, but then you know what to do next — either you follow the original or the new field). So the time cost is the same additive $\log t$ as with full path copying.

Now we're going to look at an amortized analysis of space cost to argue that we don't get the bad side of path copying. We're going to use the same general pattern of developing a potential function to measure the deferred work that we have not done, but that has built up that we're going to have to do; we're going to argue the only way for that potential to get large is many operations.

In this analysis, work no longer means time; we're actaully going to be asking to amortize space used per update operation (reads don't cost space). What's the amortized space cost for making a change to the field?

For this, we want a good potential function; we want this to be large when there's lots of work that should be done that hasn't been done yet. What would tell you there's a lot of work that's about to happen? Filled extra fields seem to be the signal for work — if you've got a full extra field, then the next time you make a change, you'll have to allocate more space. But if we think about all the full fields, the potential never drops — you don't get any energy out as you perform operations on the structure.

To get a better potential, a lot of full fields can't possibly generate work in the future, because they're dead. So we define

$$\varphi = \#(\text{live full nodes}),$$

where *live* means that the node is reachable from the current root (i.e., it can be reached during an update). (One reason we're doing partial persistence is that full persistence requires a much more complicated story — there everything is reachable, so how can you measure potential?)

Now we're done — in advanced data structures it's all about pulling out a potential function out of a black hat, and everything just works. Here

$$\text{amortized}_i = \text{real}_i + \varphi_i - \varphi_{i-1},$$

and we conclude that

$$\sum a_i = \sum r_i + \varphi_n - \varphi_0,$$

which means $\sum a_i \geq \sum r_i$ as long as $\varphi_n \geq \varphi_0$; and this is true of our potential function because $\varphi_0 = 0$ and $\varphi_n \geq 0$ (you can't have a negative number of full nodes).

So now we just need to figure out, what *is* the amortized cost? Imagine that an update happens; you go to a node and make a change. What do we do? Well, there's two cases. Case 1 is where the extra field is empty. If the extra field is empty, then the real cost is $O(1)$ (we just do a few actual writes), and the change in potential is 1 (we increment the potential by 1), so that's $O(1)$ total.

The other case is that the extra field is full. Then we still have $O(1)$ real cost from the work of copying nodes, but we decrease potential because we kill the fat child and replace it with a skinny child. We need to set the constant on the potential function to cancel out the $O(1)$ in this cost; that means when the extra field is full, the copy is free. In this case, we may have to cascade to update the parent, but the same argument holds — if the parent's also full then it's free as well. Every node on the cascade is full and therefore also free, until we get to the last node on the cascade; that's not full, but it's just one node, and so we only pay $O(1)$ cost for the single node. So the total cost over the whole cascade is $O(1)$.

So we now have a method for persisting a data structure that uses $O(1)$ space per update (this doesn't rely on any balance condition — it works as long as it's a tree).

> **Remark 3.3.** To generalize beyond trees, you use a number of extra fields that's equal to the potential in-degree of the node. The point is that with path copying you'd have to copy all the parents at the node — so if there's many things pointing at it, copying the node becomes more expensive. So you use extra fields in the node so that you don't copy it as often. And then this method works to get persistence for any data structure with constant in-degree.

> **Remark 3.4.** What's going on is we kind of folded the binary search in the timestamp log into the binary search going on in the tree itself.

§3.2 Logistics

NB is a research project from David's group; it's a tool for letting people talk about stuff online in-place (e.g., you can ask your question about the homework on the homework). Piazza is available for general logistical equations like when the exam is (there isn't). There is a small misconfiguration in SSL causing some complaints, but it should be working fine by tonight. If we have trouble logging in, we should let the staff know (it's where they'll answer questions and provide answers).

OH will be in the 5th floor lounge of Stata, since we still don't have a room.

§3.3 An application: planar point location

This is a computational geometry problem (or data structure). Computational geometry involves geometric problems, which often arise in computer graphics, and we'll look at it later.

This one is pretty natural — you're given a polygonal subdivision of the plane, and you want to build a data structure that allows you to be given a query point and you respond quickly by saying which polygon the query point is in. The polygons can be anything (they don't have to be convex, or they can even be open).

For example, every time you click the mouse, your computer has to figure out what you clicked on; that's a planar point location problem.

This is not entirely an unknown problem — if you think about the one-dimensional version of this problem, you've got a line and some division of the line into intervals, and you get a query point (a number) and want to know which interval it's in. This is binary search (you can make a binary search tree or array of the endpoint, and then solve this in $\log n$ time). Many problems you see in computational geometry are higher-dimensional generalizations of problems you've already seen. (You can imagine this in 3 dimensions as well, where you have polyhedra; in super high dimensions this has machine learning applications.)

How do we solve this? There are lots of special purpose solutions, but we'll see a solution that applies persistent data structures, and highlights a common technique in computational geometry, which is *dimension reduction*. We understand the 1D version very well, so if we can reduce the 2D version to a 1D one, then we know what to do.

And a particular technique for going from 2D space to a 1D problem is by thinking of time as just another dimension — instsead of thinking of x and y, we think of t and y. So a query point consists of a time and y, and we want to know what polygon you're in at a particular time.

And if we think about a particular time (corresponding to a vertical line), then we've just got a one-dimensional problem — the polygons divide your vertical line into a number of segments, and we just need to figure out what segment you're in, which we can do over binary search (we consider segments that intersect the line, and want to know if I'm above or below a segment; that's just a bit of elementary algebra, in place of comparisons).

So to do this, we just need an ordered list of the segments of the drawing that cross our vertical query line. You may worry there's an infinite possible number of times, but there's really only a finite number of possibilities — there's a finite number of points in the polygons, and you only care about those points. (The state of affairs only changes at the vertices of the polygons.) So we can go to every vertex of the polygons and drop a vertical, and this is going to divide our polygonal subdivision into a collection of slabs, and in each slab, there is a fixed set and order of segments to allow us to do binary search.

How many of these slabs are there going to be? There's as many slabs as there are vertices; you could use that as your primitive, or say each vertex is the intersection of a couple of lines and each line corresponds to two vertices, so the number of vertices is at most $O(\#\text{segments})$.

Now we need to make a binary search tree on each slab, where the keys are segments and we use above/below queries (when searching the tree) rather than greater-than/less-than queries. If we use n for the number of segments, then we'll need n binary search trees, which is going to cost us n^2 space; this is kind of a lot.

So what we do instead is using persistence — the change in the contents of the tree between two slabs is just a single update, and we gave a hint by calling this dimension time. Imagine a 1-dimensional person living on a line as time evolves — at any particular time you have these points, and sometimes they collide or split into more points. But we think of time moving forwards, and every time we hit a vertex, there are some changes in the state of the data structure — we hit the end of a segment and it gets removed, or we start some segments and they get added.

So we'll instead have one *persistent* binary search tree. We slide our line from left to right, remove segments from the BST as they exit to the left, and add segments to the BST as they enter from the right.

Let's understand the space and time costs for this persistent BST. How big is the tree at any given time (forgetting about persistence, for starters)? It's at most $O(n)$ (if we have n segments). So now we just need to think about the added cost in space that arises because we are making things persistent.

How many update operations do we perform? We perform $O(n)$ updates — two per segment (once when it comes in, and once when it leaves). And what can we say about the space-time cost per update? One update to the BST may involve multiple primitive reads and writes, so we have to think about all of them.

This depends on our implementation. For example, let's think about red-black trees (sometimes they teach this and sometimes they don't — recently they taught AVL trees). These have a red-black bit per node, which is used to guide the rebalancing. You walk down the tree and insert the node; then as you walk back up, you use the red-black nodes to guide rotations. It's a feature that you may update all the red-black bits along the path, but you only make one rotation (per insert or delete).

The red-black tree is balanced, so there's $O(\log n)$ nodes on your path. So we may need to make $O(\log n)$ changes to fields in the red-black tree per update, which means we're going to use $O(\log n)$ space per update (and $O(\log n)$ time, but that'd be true even without persistence). So this is in total going to cost us $O(n \log n)$ space for the persistent data structure.

And then if we want to do a query, we start by binary searching for the slabs (which is equivalent to binary searching for the right root time); then we identify the slab, and descend the binary search tree associated with that time. So that'll still take us $O(\log n)$ time to answer each query. So even though we're in two dimensions instead of one, we've achieved the same asymptotic performance (the dimension shows up multiplicatively, where we perform two binary searches — for time and vertical — instead of 1).

But actually, we can be a bit smarter. What do we need to persist? We need to persist the shape of the tree so we can find nodes. But the red-black bits are only used to tell us how to rebalance; once we've finished building the tree, we don't need them anymore. So in fact, the red-black bits don't have to be persistent. And since we only do one rotation per insert or delete, that reduces the cost for persisting an insert or delete to $O(1)$ space, rather than $O(\log n)$. And this reduces the size of the data structure to $O(n)$. So this is actually a perfect data structure — you can't do better than this (the space is proportional to the size of the input, and the lookup time is just as good as one dimension).

> **Remark 3.5.** Is this specific to red-black trees? — it seems hard to do a deletion with only $O(1)$ pointer changes. There are other binary trees that don't do lots of rotations per insert or delete. But there are also binary search trees that may do lots. We'll actually see one of those next; so that would not be a good choice for this application because of the number of primitive updates it makes on basic operations.

> **Remark 3.6.** David will mention that this particular lecture allowed him to demonstrate the utility of this class a few years earlier — students who took the class did their project on persistent operating systms (directory structures are trees, and if you want to change your file structure, you need to do this; they used persistent data structure techniques to make all that efficient in storage).

> **Remark 3.7.** On Wednesday we'll talk about what David thinks is the most magical data structure ever invented, so we should definitely come so that we can hear about it.

§4 September 11, 2024

Today we'll see what David considers one of the most beautiful and amazing data structures he's ever seen, called *splay trees*. Splay trees were developed by Sleator and Tarjan. These are for the very prosaic problem of maintaining a binary search tree, for doing the usual operations — we want to insert, delete, and search. (They also support a few other operations, but for most of this lecture we'll focus on search.)

§4.1 Binary search trees

We know binary search trees are appealing, but what goes wrong is they get out of balance; there's tons of different implementations that find some way to preserve balance (red-black trees, AVL trees, 2-3 trees, and so on). These other techniques generally try to maintain balance via some auxiliary information — marked bits or colors or things like that which tell us where the tree is in a bit of trouble, and when we do our next insert we have to fix things to stay balanced.

This is not really in keeping with our procrastination mantra — we're doing all this work to carefully balance the tree, whether or not you're going to search for things in the part of the tree you're working on rebalancing. If we're really going to be lazy, we shouldn't do anything until a search actually happens; and when we find that search is making us do lots of work, that's the time to take advantage of the work to simplify the structure.

Sleator and Tarjan figured out how to do this with *no* extra information — a splay tree is just a binary search tree without any extra information at all. The way they achieve the performance we want in a binary search tree just has to do with the clever way they adjust the tree — these are also known as *self-adjusting binary search trees*. So this performance is achieved entirely by being clever about restructuring the tree as you do searches.

And this gets way more than just $O(\log n)$ search time — they do much better than just about any imaginable data structure in lots of ways we'll see at the end of the lecture.

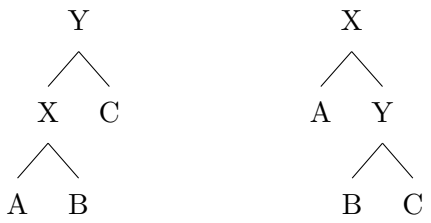
§4.2 Some motivation

Let's dive into the intuition behind splay trees. The analysis of splay trees will again be by a potential function, and hopefully we'll understand where it came from by looking at this motivation. Interestingly, there's nothing in the tree (e.g., marked bits) keeping track of this potential; the potential is just based on the shape of the tree.

> **Question 4.1.** What goes wrong in a binary search tree if you don't work hard to keep your tree balanced — why do you run into bad runtimes?

If your search took a long time, why? This means the thing you were trying to find was too far down the tree — i.e., your path was too long.

So if the path to your item is too long, and that's making your search take a long time, what should you try to do when the search takes a long time? One thing that's natural to think about is, maybe we should move this item up. How do you move an item up in a tree? The answer is rotations, which will play a role — this is a standard operation for many balanced trees.



So with rotations, you can take an element up a tree; but that sends other elements down.

So if it takes you a long time to find X, maybe you move X to the top of the tree. Then you can find X quickly. But then the adversary is going to call for something that's *now* at the bottom of the tree — there's

still a long path, so they'll ask for whatever's at the bottom. So if you keep having a long path, then the adversary can keep hammering at the path and make you pay over and over again.

So what you actually want to do is take that really long path and make it shorter. Decreasing it by 1 doesn't really help you (then you get $n + (n-1) + (n-2) + \cdots$). But if we're more aggressive in shortening, maybe we can cut the length of that path in *half*. Then what happens the next time the adversary asks you for something? Then it only costs half as much, then a quarter; so if you do this geometric decrease, then no matter how many requests come in, the total cost is going to be small.

So our fix is going to be to cut the path length in half. This will fix the problem of accessing the path; but this could have consequences elsewhere in the path that we might not like, so we'll have to be careful.

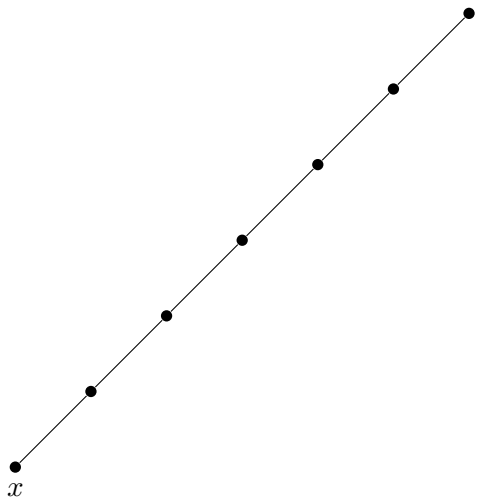
Another idea about thinking about what goes wrong is to look more directly at the notion of balance. If it's taking you a long time ot search in a tree, this means the tree is out of balance — in other words, on various nodes, the vast majority of their descendants are on one side. One definition of a balanced tree is that for any node, neither children has more than 2/3 of its descendants; then the longest path you can find is $\log_{3/2} n$, since each time you go down to a child, you're cutting the size of the subtree by a factor of 2/3.

So if your searches are taking a long time, this means child subtrees are too big compared to their parent subtree. Again, this suggests that in order to fix this, when we find this kind of balance violation, we should rotate to sort of even things out between the two sides of the subtree. More generally, if we have a very large subtree deep in our tree, then we want to raise it higher in the tree, to be at the height where it belongs given its size.

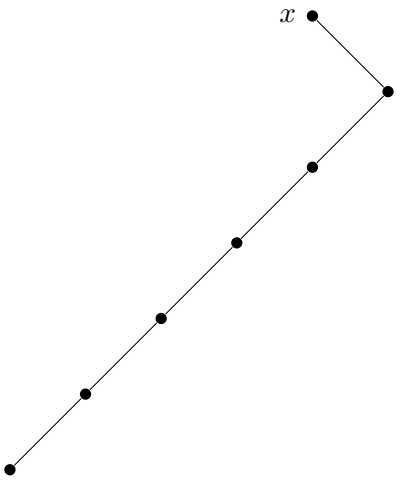
§4.3 The splay operation

Splay trees are based on a single operation (all the things we want to do with search, insert, and delete rely on this one operation), called a *splay*. This is a more carefully thought out kind of rotation.

First, why do we need to be thinking carefully about rotations? (We'll flesh this out more in the homework.) As a way to think about splaying and why we need to be careful, let's consider the case of a long path.



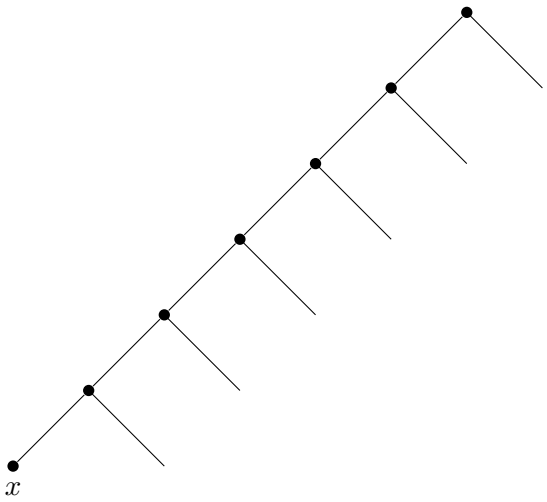
Suppose we found a long path and we want to fix this by rotating x to the top of the tree. The first rotation is going to flip x and its parent. Now we're again going to try to do a rotation, and it's the same shape as before. And in the end, we're going to be left with a tree shaped like this:



So we haven't really improved our tree shape at all.

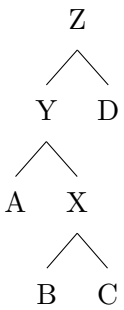
Still thinking in terms of shapes, what kind of shape could emerge from this line that would significantly shorten the path? You could split the line in two and have it branch out in two branches ('scapegoat trees' do this), but that's very global; we want to do something *local*.

So what we're going to do is try to create a shape where there's still this spine, but it's only half as long, because we managed to pull out a little bit of the tree at each step along the path.

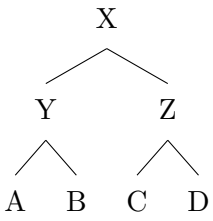


We know single rotations don't work, but we're going to use *double rotations* that do work. These come in various flavors, because we need to think about different shapes. The idea is we essentially want to move X up two steps up the tree at once, instead of just one.

Suppose we have a shape like this, called a *zigzag* double rotation (since we're moving in two directions).

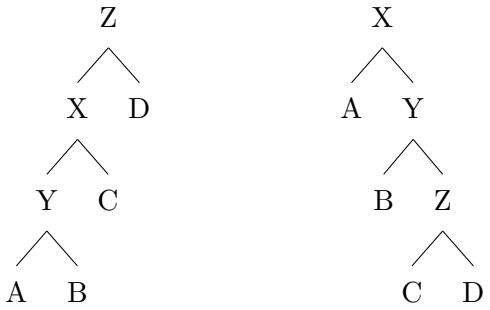


Then we change this to:



This is a local operation — it doesn't care about what's above or below, only these ndoes.

Meanwhile, when X and its parent are in the same direction, we use a *zigzig* double rotation. A *zigzag* can be implemented by two single rotations; this can be implemented by first rotating Y and then X.



The *splay* operation is basically that `splay`(x) simply uses double rotations to move x towards the root; and if its depth was odd, we do one additional single rotation to move x to the root. So that's the splay operation — we just move x to the root of the data structure in this way.

Then for `search`(x), we find x in the usual way, and then splay it to the root.

That's all — we don't have any marked bits or anything, just every time we find a node, we move it to the root using double rotations. And it turns out this is enough to make splay trees perfect data structures.

That's it for `search`; we'll talk about `insert` and `delete` later.

§4.4 The analysis

We'll do the analysis in a pretty general form, because it'll allow us to prove cool extra results later.

We're going to assume each node x has a weight w_x. (We're going to use 'node' and 'key' interchangeably; when you insert a key it goes into a node, and it'll stay with that node as you do splaying.) We'll assume $w_x \geq 0$.

We then define the *size* $s(x)$ as

$$s(x) = \sum_y w_y$$

where y ranges over all descendants of x, including x itself (so the total weights of the nodes in the subtree at x). And we define the *rank*

$$r(x) = \log s(x).$$

If we think about the role that log plays in a balanced binary search tree, this is kind of the 'height' that x *should* have (how far above the leaves it shoud be, given the weight of its descendants).

(The logarithm is base-2; this will be important for our analysis.)

We then define the potential function

$$\varphi = \sum_x r(x).$$

This is the 'where did that come from?' moment. When we looked at Fibonacci heaps and persistent data structures, the potential functions were pretty natural; David has no idea how they came up with this one, though he'll try to do some rationalizations after the fact. When he moved into his Stanford grad school office, his desk had pads full of little trees with rotations in various ways; he thinks he got Sleator's desk and saw the leftovers from him trying to figure out this potential function.

What do we want in a potential function? We want it to be large when the data structure is in bad shape, and we want that when we repair the data structure and get it into good shape, we want to release a lot of potential that will pay for the work we just did.

So let's think about a bad shape of tree and ask what happens to the potential. Well, the different elements of this sum are the ranks of the different trees; if we look at the line, the line has lots of large trees, where lots of nodes have lots of descendants. So lots of nodes have large ranks, which means the overall potential is large.

On the other hand, in a nice balanced search tree, most nodes have no descendants at all, and of the rest, the vast majority have few descendants.

So the potential is at least nice in the sense that a maximally unbalanced tree has large potential and a maximally balanced tree has small potential.

Student Question. *What is the weight of a node?*

Answer. Right now we're just assuming we have weights. It turns out that they can be whatever we want them to be, and everything we prove will be true. For simplicity, we can assume all the weights are 1; then the size is literally the size, and the rank is the logarithm of the number of nodes in the subtree. But what the weights are won't matter for this analysis.

Now that we have this definition, we can get to the important access lemma, which is the only lemma we need for splay trees.

Lemma 4.2 (Access lemma)

The amortized time to splay a node x the root t is at most $3(r(t) - r(x)) + 1$.

In other words, we start with t as the root, and we splay x to replace t.

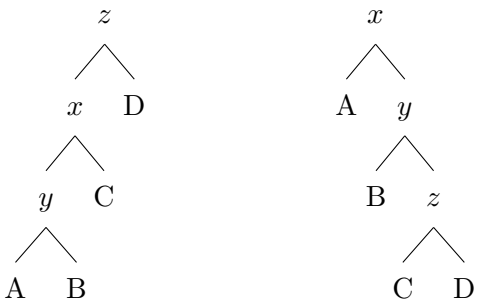
Again, let's go with the intuition where the weights are 1; then $r(t)$ is $\log_2 n$. We don't know what $r(x)$ is, but the intuition is it's how high x should be (based on its number of descendants). So this says the time it takes to splay x to the root is basd on how deep it *should* be in the tree — not how deep it *actually* is (which is the *real* time it takes).

It immediately follows that the amount of time it takes to search is at most $3 \log n$ — we have $r(x) \geq 0$ and $r(t) = \log n$, so the splay time for any node is $O(\log n)$, which means the search time is too (as search, as splay, is proportional to the path length).

Remark 4.3. To make this time bound true for *unsuccessful* searches, you have to splay the last node that you visited. (Otherwise when you search for a nonexistent node, then you wouldn't get any potential change.)

Proof of access lemma. We'll do this proof one double rotation at a time, and then telescope the sum to get the overall result — we'll show that the amortized cost of a rotation is this difference $3(r(t) - r(x))$ where t is the grandparent of x. (The $+1$ is to pay for the single rotation at the end; we'll ignore it for now.)

We'll focus on the zig-zig case, which is the harder case. (Pictures can be decptive, but if you look at the zigzag operation, it's kind of obviously making the tree nicer; and it's easier to prove that tree has the right potential. The zigzig case is more complicated.)



We'll use r to represent the ranks of nodes when we start the double rotation; and in the resulting tree, we'll use r' to measure the ranks of nodes. And we're going to analyze the amortized cost of this double rotation.

First, the *real* cost of this double rotation is 2 (measured in rotations). Meanwhile, what changes in the potential from this individual double lrotation? It's only x, y, and z that change — for A, B, C, and D, everything happening is above their heads and invisible to them. Similarly, we're not touching anything in the ancestry — none of their ranks change either. So the only changes in rank are at x, y, and z; this means the overall change in potential is

$$r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z).$$

As some intuition, we want to show that

$$2 + \Delta\varphi \leq 3(r'(x) - r(x)).$$

If we can prove this about *one* double rotation, then the access lemma follows by adding all the changes from the individual double rotations (we start with rank $r(x)$; then we do one double rotation to throw away $r(x)$ and get a new rank $r'(x)$; on the next we throw away $r'(x)$ and get a new rank $r''(x)$; and when we get all the way to the top, we have the last rank in the first position, and the original rank in the subtracted position).

It's the 2 that's the problem — the term on the right already kind of measures $\Delta\varphi$, but how are we going to pay for the 2?

The intuition is that if A, B, C, and D are empty, then what happens in the double rotation? Then φ doesn't change, and the amortized cost is 2, which is what we claimed. (This is because the only nodes that are changing in potential are x, y, and z, and in both cases you have one node with one descendants, one with two descendants, and one with three. This is intuition only, so we're pretending everything has weight 1.)

So what we need to worry about is when A, B, C, and D are not empty; and in that case the question is whether the weight is more in A and B, or in C and D.

Let's suppose that x's rank increases a lot, making $r'(x)$ much bigger than $r(x)$. Why would x's rank increase a lot? Well, this means

$$3(r'(x) - r(x)) \gg r'(x) - r(x).$$

We need one of these to deal with $\Delta\varphi$, but we're left with two terms of $r'(x) - r(x)$ to cover for the 2 term.

On the other hand, why would x's rank *not* increase much? This would mean x has about the same size as it did before; and that means A and B are really big, and C and D are very small (because previously x did not have C and D under it, and now it does, so if those were large then this would make x's rank much bigger).

So in this case, A and B are large, and C and D are small. But then we claim that the second tree has much smaller potential. The point is that by doing the double rotation, we've taken z 'away from' the large-sized trees — so z loses basically all its potential. And that is going to pay for the 2 in the actual work; this will mean this double rotation is free.

(When we talk about intuition, it's useful to focus on weight 1; but the analysis will work for any weight.)

Now let's dive into the actual analysis. The amortized cost is

$$2 + \Delta\varphi = 2 + r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z).$$

We can already start simplifying this a bit — we can cancel some terms. What can we cancel? Well, we know $r(z) = r'(x)$ — z was the root, so its rank was the log of the size of the whole tree, and in the end x is the root, so its rank is what z's used to be. So we can simplify this to

$$2 + \Delta\varphi = 2 + r'(y) + r'(z) - r(x) - r(y).$$

Now we're going to simplify things by reducing the number of letters. We've got $r'(y)$; and y is a descendant of x, so $r'(y) \le r'(x)$. Similarly, on the left we have $r(y) \ge r(x)$, since y is above x (here we're subtracting $r(y)$, so we want a lower bound). So we have

$$2 + \Delta\varphi \le 2 + r'(x) + r'(z) - r(x) - r(x).$$

And we can write this as

$$2 + \Delta\varphi \le 2 + (r'(z) - r(z)) + (r'(x) - r(x))$$

(just gathering things up a bit). The reason we did this is that $r'(x) - r(x)$ is in the result we want — we're trying to prove the amortized cost is at most $3(r'(x) - r(x))$. And since this is already here, it's sufficient to show

$$2 + r'(z) - r(x) \le 2(r'(x) - r(x)).$$

We can move some stuff around, and what this means is we want to show

$$r'(z) + r(x) - 2r'(x) \le -2.$$

Now let's figure out what these quantities actually are. Here we have $r'(z) - r'(x)$, and $r(x) - r'(x)$; let's write it out that way, so we want to show that

$$(r'(z) - r'(x)) + (r(x) - r'(x)) \le 2.$$

And now let's go back to the definitions. We defined $r'(z) = \log s'(z)$ and $r'(x) = \log s'(x)$; and we're subtracting, so we want

$$\log \frac{s'(z)}{s'(x)} + \log \frac{s(x)}{s'(x)} \le -2.$$

And what defines tehse quantities? Well, $s'(x)$ is the size of the entire final tree. And what determines $s'(z)$ and $s(x)$? First, $s'(z)$ comes from z, C, and D; and $s(x)$ comes from x, A, and B. These are two disjoint subtrees! And together, they make up everything in the tree except y. We're taking these two disjoint subtrees and asking what fraction of the whole is the first disjoint subtree, and what fraction is the second. So we can write this as

$$\log \frac{s'(z)}{s'(x)} + \log \frac{s(x)}{s'(x)} \le \log a + \log(1 - a)$$

for some a (corresponding to the relative sizes of these two parts). And this is $\log_2 a(1-a)$; we're trying to show this is upper-bounded by -2, and we can do this by determining the maximum possible value of this over all a. That maximum is at $a = \frac{1}{2}$, where the value is $\log_2 \frac{1}{4} = -2$. And the analysis is complete. □

Student Question. *Why did it have to be at most* $\log a + \log(1-a)$*?*

Answer. Through these manipulations, we transformed the quantity we wanted to bound into

$$\log \frac{s'(z)}{s'(x)} + \log \frac{s(x)}{s'(x)}.$$

These are sizes of subtrees, and we walked over to the diagram to see what subtrees we're taking the sizes of. The tree defining $s'(x)$ is the entire tree (that's the sum of all the weights); $s'(z)$ is the sum of weights of z, C, and D; and $s(x)$ is x, A, and B. And these parts are disjoint — to get the size of the *entire* tree, you just add these and y. And so we've got two fractions of the whole; so they sum to at most 1.

Student Question. *What if* y *has really large weight?*

Answer. That actually helps us — it just makes our bounds better. You can look at this in the bounds involving y. But also intuitively, when y has really large weight, you're taking it out from under z, and so z is losing a lot of potential. Granted, x is gaining that potential back, but the final potential of x actually shows up in our amortized time, so it's okay if x's potential is growing; what matters is that we're gaining this back from z. (And in the final step, if y is large then our two fractions sum to *less* than 1, and that only helps us.)

So we've now proven the access lemma, which says that the cost is just the log of what the height of x *should* be; in particular, plugging in weights of 1 tells us that the splay time and therefore search time is $O(\log n)$.

The analysis allows us to take *arbitrary* weights; and just by taking the weights to be 1, we got optimal search time. We'll now see all sorts of fun things we can do by considering other weights.

Remark 4.4. If we're grading, we've been emailed; we should meet here after class.

At 5pm today in this room, there will be a lecture by Avi Widgerson. He's one of the greatest theoretical computer scientists in the world; recently he was awarded the Turing award (which is basically the equivalent of the Nobel Prize in computer science). He's responsible for some of the most important breakthroughs in TCS, lots of which focus on complexity theory. For example, he contributed to the development of the PCP theorem, initiated some work on expander graphs and understanding the role of randomization in algorithms. He is amazing, and from personal experience of David, he is a wonderful explainer. This lecture is a broad-audience lecture, and we should know enough to understand it (the talk says no special background is necessary). What he'll be presenting about is — he spent lots of time reading Turing's old papers, and found Turing discovered lots of things we don't know he discovered. So David strongly recommends it.

§4.5 The final and initial potential

We jumped over a detail when talking about the amortized analysis. When doing amortized analysis, we can say

$$\text{amortized cost} = \text{real cost} + \varphi_{\text{final}} - \varphi_{\text{init}}.$$

We haven't talked yet about the final and initial potential. For example, when we said the weights were 1, we've already talked about the real and amortized costs; but what can we say about the final and initial potentials? Well, a potential is a sum of ranks, and ranks are logs of sizes; so each rank is going to be at most $\log n$, and there's only n nodes. So the final potential is at most $n \log n$. For the intiial potential, again that's a sum of ranks; the smallest possible rank for a node is 0 (its size is at most 1, so rank is at most 0). So there's an extra $n \log n$ term in the amortized cost. This means if you only do one operation, you won't get a good bound, since you have to incorporate this term.

More generally, if we write $W = \sum w_x$, then we can observe that $\varphi_{\text{final}} \leq n \log W$, while $\varphi_0 \geq \sum \log w_x$ (every node has its own weight in its size). What that means is the 'correction factor' is

$$\sum \log \frac{W}{w_x}.$$

If we look at this for a minute, $\log W$ is the rank of the whole tree, and from our intuition that rank is how high you should be, this is kind of the cost of splaying x to the root. So the correction factor in relating the real and amortized cost is basically the (amortized) cost of splaying each node to the root *once*. This is not surprising — if you start with a tree already there, with some giant weights at the bottom, then some operations are going to be expensive. But once you splay every node once, everything is clean and you can make the appropriate arguments.

If you start with an empty tree (we'll see inserts and deletes soon), then you'll automatically be splaying each node to the root once, and you don't have to worry about this correction factor.

§4.6 Some more applications of the access lemma

We've already seen that a splay tree is better than a balanced binary search tree — it only takes us $O(\log n)$ time to search.

§4.6.1 Static optimality

The idea here is, suppose you knew a bit more about the searches — specifically, that some nodes are going to be really popular — and you wanted to build a search tree that would be really efficient on those popular searches.

Pre-splay-tree-land, as an algorithm designer, what would you do with this — how would you adapt the structure to be very efficient for those? Popular things should be moved close to the root; then their paths will be short, so their access times will be short.

There's an interesting connection between how you should lay out that search tree and information theory, or entropy — you can think of a path down the search tree as a description of the item, so the search tree gives you a way of naming every item. And entropy is the study of how few bits you need to name things, on average. There's a famous theorem in information theory:

Theorem 4.5

If you have probabilities p_x on your items x and you want to generate a distinguishable name (in bits) for each item, then the number of bits you need on average is

$$-\sum p_x \log p_x.$$

Basically, if an item is accessed a p_x-fraction of the time, then you should use $-\log p_x$ bits for the name of that item; then this will be the average number of bits that's needed. This shows up when you're compressing files — how should you represent each token? Well, popular tokens should get short names, so that the overall size is small.

You can map this back to data structure land in terms of how deep an item should be — if it's accessed half the time, it should be the root. If it's accessed a quarter of the time, then it should be a child of the root; if it's an eighth then it should be at level 3, and so on. Then you end up with this expression as the average depth, or the average search time.

So we've just analyzed the best possible layout of a search tree when there are popularities for items. How well does a splay tree do compared to this? Well, let's go back and set $w_x = p_x$. Now $W = \sum p_x = 1$. Then the

access lemma says that the amortized time to find and splay x is equal to $\log$ tree size $-\log$ subtree size at x. And x has at least itself in its subtree, so this is at most $\log \frac{1}{p_x} = \log p_x$; so this is exactly the optimal thing (asymptotically).

It's even weirder than this — no one told the splay tree what the p_x's are. We needed p_x's before to decide how deep to put items in the tree, but the splay tree just put them there without being told; this is bizarre.

§4.6.2 Fingering

The idea of a finger in a binary search tree is that suppose a whole bunch of searches are happening in a part of the search tree. We're then instead of starting at the root, put a finger at a particular point in the tree, and then start the search from there. We may have to go up or left or right. And if you start where searches are aiming, the point is that it should take you less time, because you don't have to take the path down to the finger.

If you imagine putting a finger at node f in your search tree, then the search time is going to be approximately

$$\log(\text{difference in rank between finger and target}).$$

It turns out splay trees match this performance, even though you don't tell them what the finger is that you imagine using — you just set the weights to

$$\frac{1}{1+(\text{rank difference})^2},$$

and the access lemma gives this bound — the overall weight of the tree looks like $\frac{1}{1} + \frac{1}{2^2} + \cdots$, which is a convergent sum and therefore constant; so then the amortized cost is $\log$ constant $- \log w_x$.

So splay trees match the performance of a fingered tree, without being told what the finger is.

§4.6.3 Caches

Imagine that you've got a giant search tree, but you've got a small cache, and you cache items that have been recently accessed; and you keep a small binary search tree to keep track of these. So if you have locality of reference, then all your accesses are quick. Specifically, the access time for x can be brought down to

$$\log \#\text{distinct items accessed since the last access to } x.$$

To make this work, you create a hierarchy — you have your tiny tree, your medium-sized tree, and so on, and you esarch in increasing sizes and move things around.

It turns out splay trees match this performance; this makes sense because if you access x recently, then it's close to the root and hasn't been pulled down.

> **Remark 4.6.** When you prove this, your weights have ot change with time (which makes the proof more complicated).

§4.6.4 Final comments

And all these results come from fiddling with the weights, but the splay tree doesn't know what the weights are; so it meets all these bounds at the *same* time. (The splay tree knows nothing about these weights; it's happily going on its way and achieving all these bounds without even trying.)

We'll also see in the homework that if we access the items of a tree in order, in a normal binary search tree if you know this you can do an in-order traversal and that takes $O(n)$. And in fact, splay trees achieve this too!

Conjecture 4.7 (Dynamic optimality conjecture) — If you can do some kind of performance with binary search trees (with whatever kind of balancing and planning you want), splay trees match that.

§5 September 13, 2024

Last lecture, we talked about splay trees, which are a very incredible data structure. But we only talked about them in the context of search. You may also want to update your data structure with inserts and deletes; so now we'll finish up splay trees by talking about those operations.

§5.1 More about splay trees

§5.1.1 Insert and delete in splay trees

What's the obvious way to handle insert and delete in a splay tree, and what details might we worry about?

We could do a normal binary search tree insert, where we walk down the tree, find where the node belongs, and put it there. Does this meet our goals of providing an optimal data structure? If we just use the generic insert from BSTs, then your adversary (or your advisor) might give you a series of insertions that create a chain; if you insert these one at a time, then the last $\frac{n}{2}$ are going to take $\frac{n}{2}$ time to insert. So you don't *just* want to insert. But if you insert and then splay, then this fixes things just as with search — if you do end up taking a long path after insertion, then the splay ends up taking care of it.

But there's a little detail we need to worry about in this approach — we have to be careful in our accounting. Specifically, you have to be honest about your use of the potential function. If we say our insert walks down, attaches the node, and then does a splay, well, we know that the *splay* is paid for. But we also have to think about the attachment of the node and its impact on the potential function. When we add a node at the bottom, all its ancestors gain a descendant, which means we're adding potential to the system just by attaching the new node.

You have to worry about this, but it turns out that it's fine. We're not going to prove it because we'll see a more elegant way to do insertions. But inserting as normal and then splaying *does* work; you just have to check that the change in potential of adding a new node is okay.

What about delete? One thing you could imagine is we delete as normal, and then splay the parent. Does this work? Well, the splay of the parent pays for getting to the parent — it costs you just one more. (We haven't covered the cost of going from the parent to the child, but that's $O(1)$, so that's fine.)

Remark 5.1. This is actually more complicated if your node has two children; you'll have to move those children out in some way. But we're going to see a nicer implementation of delete soon, so we won't go through the details.

Are there issues with the potential here? Well, we can only decrease the potential, because all weights are nonnegative. So there isn't even an accounting problem here.

That's one way to handle updates; there is this annoying issue of accounting for the potential when you do an insert. So rather than doing that annoying detail, we'll see a different way of doing insertion and deletion by introducing yet other amazing operations that splay trees can do easily — specifically, *split* and *merge*.

§5.1.2 Split and merge

First, `split`(T, x) takes as input a tree T and node x, and outputs two trees — T_1 consists of all the nodes $y \leq x$, and T_2 consists of all the nodes $y > x$. We'll also look at its complement `merge`(T_1, T_2), which takes

two trees such that all of T_1 is less than all of T_2, and outputs their combination (i.e., a tree containing all the nodes). These seem like useful and natural operations; we'll talk about how to implement these, and then we'll show we can implement insert and delete using these.

First, how would we implement `split`? For this, we can just splay x and then cut off its right subtree. And we're done — this is an operation that you'd never think about doing on regular binary search trees, but for this it's trivial.

And for `merge`, we splay the maximum element of T_1; then this will have no right subtree, so we can attach T_2 as its right subtree.

Does it matter that when we're splaying $\max(T_1)$, we don't yet know what it is? We can find it easily — we just walk to the right until we get stuck, and that's the maximal element.

Now, how do we insert a node into T if we have `split` and `merge` implemented? We can split T at x, giving us T_1 and T_2. Then we can do

$$\texttt{merge}(T_1, \texttt{merge}(x, T_2))$$

(where we think of x as a tree by itself). Even more straightforwardly, you could just make T_1 the left child of x and T_2 the right child of x, and you'd be done; but it's nice to do this purely in terms of split and merge.

And for `delete`, we splay x, then remove the two subtrees T_1 and T_2, and merge them. You could also describe this purely in terms of `split` and `merge` (you split at x; then it's the minimum element, so you can remove it easily and then merge).

These operations are straightforward and natural; now let's think about their cost. They're basically based on splays, but we have to be honest about accounting for the potential.

When we do a `split`, what happens to the potential? The splay by itself is paid for (we've already analyzed that); we have to worry about what happens when we cut off the subtree, but that only *decreases* potential, so that's fine.

With `merge`, we have to worry a bit more, because we're taking two trees and making one a child of another. But what's nicer about this than `insert` is that *only* one node is gaining rank, namely the root, and so the change in potential is bounded — it's at most $\log W$ (where W is the total weight). If all the weights are 1, then we're adding $\log n$ to the amortized cost, but that's fine — `split` and `merge` are both $\log n$ time operations anyways.

§5.1.3 Some imperfections of splay trees

So that completes the story of splay trees and their amazingness. They're amazing, but they're not quite perfect. First, would they be the obvious thing to use in all your implementations requiring a predecessor-successor search, or do they have flaws?

For one thing, in some applications you don't want to be amortized — if you're playing video games, you don't want the screen to freeze for 3 seconds while it catches up on all the work it was deferring.

Another drawback of splay trees, compared to all the other tree structures we've seen, is that they change even when you're doing reads — and they *have* to. So a read of the splay tree causes writes. And that's bad — you don't want to do unnecessary writes, because this does weird things to your caches.

This also creates a problem if we think about persistent data structures. If you wanted to make splay trees persistent, what would happen? Every time you did a read you'd have to do a splay, and every splay would require you to make changes to the nodes, which would have to be persistent. We showd last time that the amortized cost of a read is $\log n$, which means you're changing $\log n$ nodes on each read, so you're adding $\log n$ to your storage every time you do a *read*. So that's not good; the data structure is going to eat more and more of your memory.

So there's two small problems. Some of this you can try to fix using various heuristics. One is to only splay on *long* searches — splaying is an optional activity. If I don't splay when my search is fast, that's okay, because my search is fast. If I do splay when my search is slow, that's also fine, because my change in potential pays for that. So this is valid and reduces the amount of writing that you do.

Another option is to stop splaying after a 'while.' (These are heuristics, so we're not being very precise.) The idea is that we said the splay tree somehow figures out how to put frequently accessed stuff at the top. If you have constant probabilities for items, then the splay tree will eventually put the popular items at the top, and then you can just stop splaying — the tree will have figured out the statically optimal structure.

§5.1.4 Compression

Splay trees can do data compression. If you want to send a message, there's interest in compressing it (using fewer bits). There are many compression algorithms, and splay trees give you one. You can take your alphabet and put it into a splay tree. Now any time you want to send a letter, you look for that letter in the splay tree. This gives you a series of L/R (or 0/1), and you can treat those as the encoding of that letter. If something is very popular, it'll be on a short path from the splay tree, which means it'll have a short description in bits; so when you send popular things you'll only send a small number of things.

On the receiving end, you get this series of bits; and you keep a copy of the same splay tree, and the series of bits tells you what letter got sent. You also need to do the splay the sender did so that your splay tree stays in sync with theirs.

Because of the static optimality theorem, you can prove that the number of bits you're sending is optimal if you have a fixed distribution.

§5.1.5 Dynamic optimality conjecture

Last time we mentioned the dynamic optimality conjecture, which says if you have *any* BST that does any rotations and has complete awareness of what accesses will happen in the future and is able to do whatever rotations are best for the future accesses (we'll formalize this when we study online algorithms later) — whatever that prophetic data structure is doing, splay trees do just as well.

No one has ever been able to find a counterexample, to make a splay tree do badly compared to the optimum. But no one has proven it yet. The closest we've gotten is that a group of people developed *tango trees*, and showed these are within a multiplicative $O(\log \log n)$ of the optimum. In sequential algorithms, we think that if you've got a polynomial time algorithm, tacking on an extra log is not a big deal; so here we've got $\log n$ time algorithms, which means tacking on a $\log \log n$ is still pretty good.

Sleator then developed a 'tango-splay' variant, which kind of looks like a splay tree but incorporates the extra information used in tango trees, and it matches the same $\log \log n$ factor.

But the true result which must be true — that splay trees are in fact dynamically optimal — is still waiting out there for us to turn in on Wednesday, since it's in the optional homework.

§5.2 Introduction to indirect addressing

For the next topic, so far all our data structures have been comparison-based (Fibonacci heaps, location queries, splay trees). Comparisons are great, but they have formal limits — for data structures that only access things through comparisons, you can prove formal bounds (e.g., you can't implement heap delete-min in $o(\log n)$ time only using comparisons; and with trees you can't do search in $o(\log n)$ only using comparisons).

So now we're going to abandon that, and start using the fact that things we're working with are usually *numbers*, and we can take advantage of that. Today we'll focus on the fact that using integers gives you indirect access into arrays. This is trivial but very powerful — you can allocate this huge chunk of memory and with one instruction, tell your computer to get you something from location 973.

Next week we're going to see a way you can actually use algebra (mathematical operations on the numbers) to improve performance. But for today, it's all about indirect addressing.

§5.3 Motivation: shortest paths

Single-source shortest paths can be solved using Fibonacci heaps in $O(m + n \log n)$ time, which is fantastic compared to the $O(m \log n)$ time we arrived in this class thinking was good. But this still isn't as fast as it could be. David mentioned there's a paper showing that $O(m + n \log n)$ is optimal, but in the *comparison* model — Djikstra's algorithm is instance-optimal in that for *any* graph, there is no algorithm faster than Djikstra's for that graph using comparisons.

But the caveat 'using comparisons' is one that we can play with. Let's suppose that the edge lengths are all small (nonnegative) integers, at most some value C. In the easiest case, suppose $C = 1$ — so all the edge lengths in your graph are 1 (we won't worry about zeros for now). Then you can do better — you can do BFS, and all of a sudden, you're achieving $O(m)$ time.

Can we generalize this? If $C = 2$, then you can pretend all the weight-2 edges are actually 2 edges — we add in a midpoint of each edge, and pretend we do BFS through that midpoint. In general, this will give us an $O(Cm)$ runtime (since we've multiplied the number of edges by a factor of C).

That's an improvement for small values of C, but we're going to try to do better. The thing to think about is that in this case, the priority queue that we're using has two interesting features that are not general for priority queues. One is that it only has C distinct values in it — because when we have a certain length in the priority queue (or when we pull it out), we can only add things within C of the length we just pulled out. The point is that you don't insert anything with value $k + C$ until you delete the value k from the priority queue — in order to delete k the minimum has to be at least $k + 1$, and at this point you only have values up to $k + C$.

And the second property is that it's a *monotone* priority queue — the min is non-decreasing over time.

§5.3.1 Dial's algorithm

How can we take advantage of this? This is Dial's algorithm, from the 1970s. The idea is that instead of making some fancy data structure of keys and things, we just keep an array of buckets — we make a big array representing the possible distances that things have frmo the source, and for starters, we put in the source at position 0 (because the distance of the source s from itself is 0). Then we delete s and look at its neighbors, and add them to the priority queue in their appropriate buckets — maybe we put x in position 4, y in position 8, and so on. So we're now tracking the distances of all the nodes adjacent to x. (We hang a linked list off each entry to hold more items if they're there.)

That's the first thing we do in Djikstra; what's the next? The next thing is that we find the minimum of the things in the priority queue. And we know the next element is in the next C, so we can just find it by brute force — we can just walk forward in the array until we find the next minimum (this takes at most C steps).

So `insert` is implemented by adding to the right bucket, and `delete-min` is implemented by walking forward until you find a nonempty bucket for the next minimum.

If we use this as our implementation, how fast is our algorithm going to run? Well, we've got implementations of `insert` and `delete-min`; in Djikstra's algorithm we do m inserts, and each takes $O(1)$ time. And for

`delete-min`, we know the next key is only C away, so even for a single `delete-min`, you're only going to take C time. We only call `delete-min` n times (each time we pull one vertex out of the priority queue).

But there's a stronger argument that we'll stick with because we'll use it more — the `delete-min`s start at 0 with the source. And then we start moving forwards; and the priority queue is monotone, so nothing ever gets inserted behind me (I'm only adding values to the one I'm deleting). So as I search for minima, I keep only moving forwards in this array; I never move backwards. This means the total time for the `delete-min`s is at most the maximum index I can reach in this array of buckets. And the maximum possible distance I can reach is $D \leq (n-1)C$, since whatever the other vertices are, the shortest path between two has at most $n-1$ edges, each with weight at most C.

So that means the total time is at most $D \leq (n-1)C$, because each single advance is constant. So overall, we're achieving $O(m + nC)$, which is substantially better than $O(mC)$.

§5.3.2 Conserving space

This is still somewhat unsatisfactory; it's great on time, but in modern computers, it's often space that matters more than time. Even for a moderate sized graph, when we multiply n by C, we might end up with something huge. And we can exhaust the space in our laptop much faster than we can exhaust time (a laptop can read all of memory in a few seconds, and we want to solve problems that might take more than a few seconds).

How do we conserve space? The idea is that all elemnents we put in are within C of the current minimum, so at any point, there's only an interval of length C that contains empty buckets. So there's no point in keeping the rest of the array around.

So we simulate this giant array by a sort of spiral — where the number of elements in one turn of the spiral is C (or maybe $C+1$), so that by the time we need to put something in one place, we're no longer using that bucket for the previous value, because the minimum has already moved onwards. Mathematically, an item at distance d should be put into the bucket d mod $C+1$. You can't have two items at a distance greater than C from each other, so the value of d mod $C+1$ tells you the actual value of d (if you keep track of how much you've gone around the circuit).

So this reduces the space from nC, which is huge, to C, which is tiny.

§5.4 2-level buckets

So Dial's algorithm is $O(m + nC)$ time and $O(C)$ space, which is pretty cool. But theoreticians are never satisfied; can we do better? (Really it's space $O(n + C)$ because we have to store the nodes in the final shortest paths tree.)

> **Student Question.** *Could we use a hash table and use less than C space?*
>
> **Answer.** We'll talk about hash tables next week. But maybe the problem is that hash tables lose track of order — the nicely lined up array lets you move in increasing order of keys.

We've used this hack to reduce space, but let's concentrate on time. Where are we spending most of our time? Well, we're spending most of our time traversing this sparse array — there's lots of empty space, and every once in a while we find an item. And the larger C gets, the bigger these empty spaces get, and the more time we're wasting. So how do we fix that?

We can generalize the bucketing technique using *two-level buckets*. Let's take this bigger array, which is full of empty spots, and make batches of buckets — and have a second-level data structure that tells us whether a region is empty or not (so that's really just a bit).

If we have this summary structure on top of our array, how can that speed up our delete-min calculation? When we're rujning `delete-min`, we can ask our summary structure, is there anything in this region? And if no, then we can just jump to the next summary bucket. If there is something in the region, then we can drop down into the original data structure and find this element. (And the fact that there was something in here means we don't have to go further in the summary structure.)

So we're going to make *blocks* of b buckets, We have nC original buckets, so each block will have nC/b items. And in each summary bucket, we keep a *count* of the items in its block.

> **Remark 5.2.** We also need to think about `decrease-key`. But this takes $O(1)$ — we take out the value from its current bucket, and put it in its new bucket. This corresponds to paying m in the runtime.

Now that we're using 2-level buckets, what do we have to do on an `insert`? You add your element to the right bucket, and then increment the summary by 1.

For `decrease-key`, it's pretty much the same — you move the buckets, and update the two summaries. So these are both clearly $O(1)$.

What about `delete-min`? We first scan the summary structure to find the first nonempty block (which might be the block we're currently in, if it still has more items). And then we scan the block to find the next item.

You might potentially have to scan the whole summary structure to find a nonempty block, and then scan the block (which takes b time). This might look bad for a single operation, but what about amortized time? In fact, the *total* time it takes to scan the summary structure is nC/b over the entire algorithm. We're also using $O(b)$ every time we scan a block, but we only do this n times. So overall, the runtime for the n `delete-min`s is $O(nb + nC/b)$.

Now we should set $b = \sqrt{C}$ — we want to minimize this runtime, and we get to pick b, so we can balance this by setting $b = \sqrt{C}$ so that the two terms are equal, and we get a runtime of $O(n\sqrt{C})$ for all the `delete-min`'s; this gives us $O(m + n\sqrt{C})$ for shortest paths. So we just went from C to $\sqrt{C}$. This matters a lot — for example, if $C = 2^{32}$, then we've just made things practical that were not previously.

(You can also use the same space trick to make your summary structure and inner structure short.)

But theoreticians are never satisfied; can we do better? Well, we could add a third level — we could have a third tier of 'superblocks,' where you summarize blocks of blocks and say how many things are in each block of blocks. We're not going to do the math, but this improves the runtime to $O(m + nC^{1/3})$. And we can do better by using a fourth block, which turns this to $C^{1/4}$, and so on.

Can we keep on going until C vanishes entirely and we have an $m + n$ runtime? Well, the problem is that there's a constant hiding in this O, that's not a constant if we keep on tacking on buckets. The point is that our algorithm depends not only on the width of our aray, but also on the height — how many layers of summaries there are. If we have to walk down k layers of summaries, that's going to take us k time. So there's actually a k that shows up here.

§5.5 Tries

To take this further, we'll want k layers of summaries. But it'll be convenient to represent things differently. We'll have a *trie*, which is kind of like a tree, but where child-pointers no longer correspond to 0 and 1, but to symbols in your alphabet.

So we'll take a *trie*, which is a depth-k tree over arrays of size Δ. So we have a width-Δ array, and each elemnt of that width-Δ array points to a child which is also an element of a width-Δ array, and all the way

down for k levels (where the levels correspond to the summary structure levels, but now we think of k as variable rather than constant).

We'll focus on numbers in the range 1 to C (we talked about how the set of active numbers is only in a range of $C+1$, so if you do everything mod $C+1$ that works out; so we can do that here). And we can think of those numbers in a base-Δ representation — so we have Δ values in each. So the first level gives us the most significant digit base Δ, and so on — each number corresponds to a path in this trie to a particular leaf.

In other words, each number has a base-Δ representation that traces a path in the trie. And we want to represent all the numbers in the range 1 to C. Since we have depth k and width Δ, we have Δ^k leaves; and we want this to be C so that all the possible numbers fit into this trie. (We might actually want this to be $C+1$ just to be careful.)

Now let's suppose we're working with this trie. How long does it take us to insert a key into this trie? The answer is k — we have to walk down the proper path to find the bucket at a leaf. The same is true for `decrease-key`.

How long does it take to do a `delete-min` from this trie? At every level, we scan our bucket; if we get to the end and it's empty, then we go up a level. Then we scan this and see if I have any nonempty children; if I do, then I go down. So I start from the bottom-left and go up as long as I find empty nodes, then go over, and then go down. I go up at most k steps and down at most k steps, and in between each step I'm scanning something of width Δ, which takes Δ time.

So that means the overall runtime is going to be $O(km + kC^{1/k}n)$ (recall $\Delta = C^{1/k}$, and we're doing m `insert`s and `decrease-key`s and n `delete-min`s).

> **Student Question.** *Why does insert not take $O(1)$ — don't we know the exact position we're inserting at?*
>
> **Answer.** We *could* use an implicit trie (a binary heap is a binary tree but represented in an array) — we could associate each node of the trie with a particular position in an array of size C. But to do this well, you need information at the different nodes that you have to check. In particular, as you're going up, the internal nodes are holding summary information about whether they have nonempty children. So even if you represent the trie in an array, you would still need to check various values in order to know which values to check next as you descend the tree representation (for `find-min`). In order to support `find-min`, we need this summary informatino, and you have to *update* that summary information when doing `insert`.

And k is in our control, so we get to choose it. We've got a k in both places, so we balance by setting $m = C^{1/k}n$, which means $k = \log_{m/n} C$, and our overall runtime is $O(m \log_{m/n} C)$. And in general m is much larger than n, so this is a large base; this makes this quantity quite small, even if C is very big.

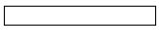
§5.6 Laziness

This data structure is pretty good, but theoreticians are never satisfied; let's make it eve better. This trick was developed by Denardo and Fox (1979). We've been focused on integers, and we've lost track of laziness; David claims we're being insufficiently lazy in the way we work with this trie. We're doing a lot of work that's ultimately going to be fruitless. Why — what's the typical behavior of an item in this trie? We insert a node somewhere in the trie, and maybe it lands in some place in the bottom. And then we do a `decrease-key`; and when we do that, well, we pick it up from here and put it somewhere else. And now we do another `decrease-key`, so we pick it up and move it somewhere else. And we keep on moving it around in the tree. Meanwhile, all the `delete-min`s are still happenign all the way on the left. So why did we waste so much energy moving stuff around in the part of the tree that didn't really matter?

The only place where it's important to know whether an element is in the rightmost block or the one before is when these are pertinent to the current min — before then, we don't need to worry about the fine-grained differences.

So the idea is that you don't push items down the trie until it is necessary for correctness. We know there's some fun stuff going on at the minimum element, which is in some block at the bottom; so we'll be paying lots of attention to that block. But items that are supposed to be descendants of something way to the right, we won't be looking at. So instead of expanding out the whole data structure from that field, we just lazily leave all the items that'll need to be in that part of the data structure at the top. Similarly, anything that belongs as a child of the rightmost guy at the top — we're just not going to make that part of the data structure, they'll just be hanging out here.

So we only maintain the blocks that are *on the path to the current minimum* — those blocks are materialized. But an element which is supposed to be inserted and walked down the tree — it goes down the tree, but *stops* when it falls off the current path to the min.



So we have an element, and it was inserted and it landed here. Eventually we run out of the current block and move to the next; we still don't care about this element. We dont' care about this element until our min-pointer gets ot htis actual field; at that point, any elements being stored in this field (or the linked list hanging off it) become possible minima. So at *that* point, we materialize the next block down.

So as we advance this min-pointer, we materialize the new blocks when necessary to take the next step. The `delete-min` finally gets to this field, and that means we have to take all the items being currently held there, and unpack them into the next block. How long does it take to take all the items in this linked list and move them to their appropriate entry the next block down? This is linear, because we have indirect addressing and indirect lookup — we just look at the next digit of this number, and move it to the correct place. So in time proportional to the number of elements, we've prepared the next block.

Now let's talk about the operations. How do we do an insert? Well, we already described that — we start at the top of the tree and walk down until you're about to fall off the cliff, and then you stop and just hang out there until it's time for you to proceed.

For `decrease-key`, if I decrease the key of an element, it's going to go left — we already know it belongs in this block (because there's no earlier block). So it just has to move to an earlier bucket in this block. And if that happens to be the materialized bucket, then it has to drop. So when we do `decrease-key`, we just move it left within its block, and then possibly down.

What this means is that one item gets inserted and then only moves downwards through `decrease-key` operations and expansion of new blocks (what we've been calling materialization). The item sits where it is; if `decrease-key` happens it moves over and possibly down; if th min-pointer reaches it then it moves down. But either way, it's only moving down after the initial insertion.

And that tells us how much time it's going to take — in fact, `insert` has $O(k)$ amortized cost. Because when we put it in, we know we're going to spend k work moving it down; we just charge all that work to the `insert`, so we don't have to charge it later.

Now let's talk about `delete-min`. Well, we scan the last block; if it's empty, we need to go up. We go up a certain distance until we find a nonempty block; then we go over that block and find the first nonempty bucket. THen we might need to materialize blocks in order to go down and find the next minimum. But we've already charged for those in the insertion, so materialization is free; so we just need to go up, over, and down.

We scanned to determine our block was empty; then going up takes time proportional to how far I go up, which is at most k; and then going down is also at most k (because there are at most k levels in the tree,

and the expansions are free). So that tells us `delete-min` just rises to the first nonempty level, scans for the next nonempty bucket (in that level), and then descends to the next minimum. And the time for the rise and descent is k, and for the scan is Δ; so this just takes us $k + \Delta$ time. But the k is already paid for by the insert that we did – the k is the distance we have to move down to the minimum, and we can charge that to the k we spent inserting the minimum. So then all that's left is Δ for `delete-min`.

SO that's the end of the story for this data structure — we get $O(k)$ for `insert`, $O(\Delta) = O(C^{1/k})$ for `delete-min`, and if we play the balancing games, we get a runtime of $O(m + n(k + C^{1/k}))$, which balances out to $O(m + (n \log C)/\log\log C)$. We don't have time to do the math, but if $C = 2^{32}$ or 2^{64} or something, then this is just $64n$. So this is going to be linear even when your numbers are absolutely huge.

There's one more story; this still isn't the best possible. Silversteen worked with Goldberg and said, how can we make this faster? Where are we taking too much time in this data structure? Well, we're scanning these blocks for the next nonempty element. If you want to find the smallest element of a list (i.e, you want ot insert things and find the smallest one), you should actually use a priority queue. So they developed HOT-queues, which is a 'heap-on-top priority queue' — at the top of the data structure you put a inary heap to keep track of the first nonempty thing. This improves the runtime to $O(m + n(\log C)^{1/3})$.

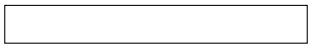
This *still* isn't the best you can do; on Monday we'll see anotehr twist of heaps-on-heaps-on-heaps that gives an exponential improvement over this runtime.

§6 September 16, 2024

§6.1 Tries

Last time we talked about using buckets to deal with integers; and we saw the efficient *trie* data structure for monotone priority queues. Last time we derived this, but now that we've done the derivation, we'll put it all in one place.

The data structure is a partial representation of a degree-Δ trie. But in order to save both space and time, it only materializes one node per level of the trie.



So you have one degree-Δ bucket for each level of the trie; and somewhere in the bottom level is the min. There is a path from the root through various nodes of the trie to the minimum element, and you can read off the minimum element using base-Δ (each square in some level corresponds to a digit).

We only represent the minimum element fully in this trie. Other elements go down as long as they're following a path to the minimum; but as soon as their digit differs from the minimum, they get placed into an appropriate bucket for that digit in whatever bucket of *that level* of the trie they should occupy. For example, if we have a number which begins with 1, but whose second digit is 4 rather than 3 (as in our minimum), then it just stops at the second level of the trie.

Now we need to talk about how to update this for insert, decrease-key, and delete-min. For insert, we simply walk down the current min path until we fall off. This doesn't even require special comparisons to the min; we just look at the first digit and check whether we're in the same expanded bucket as the current min; if we are then we go down, and if we're not in an expanded bucket then we stop there. So we just walk down, and stop in the right bucket at that level.

In terms of auxiliary information, we also keep a count at each level of the number of items *in that level* (this doesn't include anything that has gone further down, which will be important later).

How do we do decrease-key? We just take the element out of its current bucket and move it to where it should be. It's important that this is a decrease, so whatever bucket we're in, we either stay where we are or move to the left in our current level; either we move to a current non-expanded bucket and stop there, or we move all the way to the current *expanded* bucket, in which case we drop. So that's all a decrease-key is. (Of course, we also update the counts; if we don't change levels then the count doesn't change, and if we drop then we increase the new count and decrease the old count.) This means you always descend and never rise — a decrease-key never forces you into a higher level. That means over any course of inserts and decrease-keys of a particular element, the total work is at most the depth of the trie (which we call k).

So the total work per element, of the insert and *all* decrease-keys, is k. We'll charge that to the insert; then insert is $O(k)$ amortized, but there's no work left in terms of decrease-key except for working in the level you end up at, which is $O(1)$ amortized.

> **Student Question.** *Isn't there cost to walking left?*
>
> **Answer.** No, we're using array indices to tell us where this should go — we don't have to walk left. This is the power of indirect address.

The clever bit comes when we talk about delete-min. To handle this, we have to remove the minimum and find the next minimum; as with all data structures, we do this by scanning. If we scan and find a nonempty bucket in this level, that's our new minimum. The other thing that can happen is that we could fall off this array. But in that case, we don't want to do the scan — this is why we keep the counts. If the count drops to 0, we know this array is empty, so we go up to the next level. This array might also be empty (if there was only one full bucket). So we keep going up until we reach an array that's nonempty. As we scan across we know we will find a nonempty bucket. Then we expand down, opening the next level and moving every item to the appropriate level. As we do that, we also discover the smallest nonempty bucket in this level, and we expand that. We keep on doing these expansions, one level at a time, until we expand all the way down to the bottom; and then we'll have revealed the new minimum, and everything will be in its proper place.

So on delete-min, we move up to the first nonempty level (which might be the one we're already at, at the bottom). We then scan for the first nonempty bucket, and then we expand it recursively to complete the data structure and reveal the new minimum. (Unlike with decrease-key, we do have to walk right here because we need to *find* the next nonempty bucket.)

How much work does this involve? First, scanning for the first nonempty bucket takes time $O(\Delta)$, since the array has size Δ. Expanding the items that we find in the first nonempty bucket doesn't actually cost us anything — we paid for pushing those elements down when we charged k for the insert, so now it doesn't cost us anything more. What about moving up, though? We can maybe charge that to the insert for the minimum as well (we can charge $2k$ instead of k for the insert). But alternatively, the amount of time we spend moving up is the same as the amount of time we spend moving down; so the total work spent moving down is the same as the total work moving down. And we just said the work of moving down is 0, so the work of moving up is also 0 (in the amortized sense).

So all that wasn't paid for is the Δ for scanning *one* level of the trie. That's how this data structure keeps all the best parts of what we had before — we only have to do work at *one* level, instead of all the different levels (as with the layered summary structures).

This gives us $O(k)$ insert and $O(\Delta)$ delete-min; and from that, we got excellent runtimes for shortest paths. We won't redo the derivation, but we ended up with a runtime of

$$O(m + n(k + C^{1/k})),$$

and by choosing k appropriately, this is

$$O\left(m + n\frac{\log C}{\log\log C}\right).$$

David's brother's roommate managed to reduce this by putting a heap on top, to

$$O\left(m + n(\log C)^{1/3}\right).$$

That's where we finished up last time. We should mention the paper about this was not just a theory paper; there's a nice line of work in *algorithm engineering* where people actually implement algorithms and see what you need to do to get good performance *in practice*. The author did this with Andrew Goldberg (who has done lots of work on this type of thing — he starts with algorithms that have good runtime, and shows what you do to tune it to get good performance in practice).

§6.2 Van Emde Boas priority queues

Last time we also promised we're not done; there's yet another exponential improvement in the performance of priority queues. Perversely, we'll actually go backwards in time — the above structure was from 1979, and Van Emde Boas priority queues are from 1977.

The idea here is still about buckets. But it's cranking the recursion a whole new layer. We already have the idea of having a collection of buckets, and then we think about putting a summary data structure over the collection of buckets. This leads to a question of how we find the smallest thing in the summary data structure.

With the heap-on-top approach, we're finding the smallest element in an array where we're putting things in and taking things out; we used a priority queue for that. But here the approach is we should just recursively use whatever structure we just developed for doing this.

Interestingly, for lots of algorithms, exactly how you order things is just a matter of sliding the algorithm a bit. But here you have to define things very carefully to get the right runtimes.

We use the same idea as before — we have an array of buckets, with a summary structure on top, and a recursive priority queue for the summary. We actually just have one layer of summary structure; everything else is swallowed by the recursion.

A van Emde Boas queue on b-bit numbers (it's simpler to talk about the number of bits rather than the value) contains:

- A box holding the current minimum $Q_{\min}$. (We don't have a pointer to it; we just store it in a box.)
- An array Q_{low} of queues. We have a whole space of numbers, and we'll divide it into smaller parts with a priority queue for each. Specifically, we have b bits; we break them into two groups of $\frac{b}{2}$ bits, called *high* and *low*. Our low queues are queues on the set of *low-order bits* for *each* possible high-order bit value.
- A single queue Q_{high} tracking which Q_{low} arrays are nonempty.

Another way to look at this is that we have b bits, which allows us to cover integers of size up to $U = 2^b$. The Q_{high} is a queue on $2^{b/2} = \sqrt{U}$ possible high-bit values. So we're taking the universe and thinking of it as a $\sqrt{U} \times \sqrt{U}$ grid; each row gets a low-bit queue, and then we have a high-bit queue telling us which rows are nonempty.

The minimum is actually not stored recursively; we set it aside, and everything *else* gets stored recursively.

§6.2.1 Inserts

What do we need to do when we insert a new value x into the queue? The first thing we need to check is, is this a new minimum? If $x < Q_{\min}$, then we swap them; now we can assume $x \geq Q_{\min}$, so x needs to go into the recursive part of the data structure.

To do this, we expand $x = (x_h, x_\ell)$ (where each has $\frac{b}{2}$ bits). Then we use x_h to look for the queue responsible for the low-order bits that go with those high-order bits. If that queue is nonempty, then we insert x_ℓ in it, and that completes the insertion. (So we use the high bits to say which recursive queue, and then we put the low bits into that recursive queue.) So we look at $Q_{\text{low}}[x_h]$, and if it's nonempty, we just add x_ℓ to it.

Otherwise, we need to create a queue at the appropriate location, i.e., as $Q_{\text{low}}[x_h]$, with x_ℓ as its minimum (i.e., its sole element). We also have to insert x_h into Q_{high}, to say there is now a nonempty queue at x_h.

What can we say about the runtime of this operation? Well, we can write a recurrence; if we start with a b-bit number that's being inserted, then we do a constant amount of work to expand $x = (x_h, x_\ell)$; then we do a lookup, which is constant.

If we assume for now that there's always a nonempty queue that we just insert into, then we would have

$$t(b) = 1 + t(b/2).$$

The runtime for this recurrence would be $O(\log b)$. That's the exponential improvement — this is $\log \log U$ (it's logarithmic in the *number of bits* of your number, not its value).

But what about the second case? Now we have to worry because it looks like we're doing a *pair* of operations, and if we instead had $t(b) = 1 + 2t(b/2)$, then our runtime would now be b (as opposed to $\log b$). That would be pretty bad. So we don't want a 2 there.

But it looks like we have a pair of data structure operations, so why *don't* we have a 2? The point is that because we're creating a *new* queue $Q_{\text{low}}[x_h]$ of constant size, which just has a value in the minimum and doesn't have any other values, creating that new queue with its single element is constant-time. So we're still just doing a *single* recursive operation on a $\frac{b}{2}$-bit queue.

So recursively, either we're inserting in the low queue or we're inserting in the high queue; either way we're only doing one insert into the recursive $\frac{b}{2}$-bit queue.

§6.2.2 Delete-min

First, we know where the min is — we just have to remove $Q_{\min}$. But the challenge is, how do we replace it? The first thing we need to do is find the smallest collection of higher-order bits for which there is a nonempty queue. If $Q_{\text{high}_{\min}}$ is null, then Q_{high} is empty, so Q becomes empty (and we can report that). Otherwise, it gives the smallest x_h with a nonempty bucket. So how do we find the rest of the number that we need to make our new min? We then find the rest of the bits via a delete-min from $Q_{\text{low}}[x_h]$.

Now, this might empty $Q_{\text{low}}[x_h]$ — we might be deleting the last element in this queue. If that happens, then we need to go back to Q_{high} and delete x_h (i.e., run delete-min on Q_{high}, since this is its minimum element). (This is because Q_{high} is supposed to track the nonempty recursive structures, so if we empty one, we need to update this. Even if this empties Q_{high}, though, we don't have to do any more updates — we've set aside one element already, so this doesn't mean our queue is empty, it means there's one element.)

> **Remark 6.1.** It's important for this argument that we're setting aside the minimum (rather than just keeping a pointer to it and having it in the data structure).

What's the runtime for this? Well, we're doing constant work to check for emptiness. It takes constsant time to look up $Q_{\text{high}_{\min}}$ (it's in a box). Then we call a delete-min recursively; if that was all we did, then

we'd get $t(b) = 1 + t(b/2)$. But we have to worry, when we do this delete-min, what if we discover we've emptied the queue and we have to do a second delete-min? So do we need $1 + 2t(b/2)$?

But if we emptied the queue, then the first delete-min was only constant cost — before we called the delete-min there was only one element, which was in the box. So if the second delete-min happens, then the first delete-min is constant. This means we do in fact have

$$t(b) = 1 + t(b/2),$$

which means delete-min is also $O(\log b) = O(\log \log U)$. So now we've really squeezed out everything we can from working with integers — we've got a doubly exponential improvement in scanning.

§6.2.3 Limitations and issues

Thorup (1996) gave a small improvement on this. This talks about the size of the universe; if you're maintaining a priority queue of 10 gigantic numbers, this data structure will be very slow. Thorup showed you can transform this into a data structure with $O(\log \log n)$ runtime, where n is the number of items in the priority queue; however, they must be integers (that fit in a machine word). The practical implications are limited, but if you ever build a machine with 20000-bit integers, then this will let you do fast operations even dealing with huge integers. (Of course, the hardware of dealing with 20000-bit integers is scary; but becoming independent of the word size is theoretically nice. In fact, the paper shows you can transfer between priority queues and sorting, which is very interesting; you can also do that transformation between bits and number of items.)

The downside of these van Emde Boas queues is space. Just at the top level, we're creating an array of size $\sqrt{U}$, no matter how many items you're storing. Certainly at the beginning $\sqrt{U}$ will be bigger than the number of items you store; this might easily be bigger than all the memory you have. (Of course it drops off very quickly; the second level has arrays of size $U^{1/4}$, which is negligible. But you also have one of these for every item, so that gets multiplied by n — so you have $\sqrt{U}$ at the top and $nU^{1/4}$ at the second level.) So you're managing a gigantic array, which is almost entirely empty.

But this is not a new problem — if you want to do fast lookup of keys you stick them in an array, but if you only have a few keys then you have lots of wasted space.

But do you need this? If you just want to store things so that if you have an integer you can look up the value stored there in constant time (as in an array), but you don't want to use all the space, what do you use? You can use a hash table (or a hash map or dictionary). This will be our next topic — how do you efficiently do $O(1)$ lookups for equality? Now we're moving away from delete-min (where you need to look at size) and just look at equality lookups.

In TCS, some of the things we'll do to get $O(1)$ hash tables are a bit questionable in terms of their actual complexity. The way this is said is that van Emde Boas queues give you incredible runtime if you either allocate all this space or use randomization (since we don't have perfectly deterministic hash tables yet).

> **Student Question.** *What happens if you decrease-key below the current minimum in a trie?*
>
> **Answer.** This violates the definition of a *monotone* priority queue, which doesn't allow that. (This property arises a lot in practice, e.g., for shortest paths.) Van Emde Boas queues don't have this restriction. They're more powerful than you can imagine — it turns out it's not just a priority queue but actually a search tree, in that you can implement find-predecessor and find-successor in the same way.

Remark 6.2. David will not be here on Wednesday. We have a recording of David's lecture from a previous year, which you can watch at home if you want. We will play it here at class time, and TAs will be here; if you have questions they will pause it and answer questions.

We will cover the conclusion of hashing and an introduction to max-flow. We'll then dive deep into max-flow next week (Friday is a holiday). So this will be a light week for this class, especially if you've seen perfect hashing and max-flow; but it'll go back to its usual pace next week.

§6.3 Hashing

§6.3.1 Motivation

The idea of hashing is to save space. We have n items in a range from 1 to m (these are again integers — but you can think of lots of things as an integer — for example a string is just bits, so you can think of it as an integer). We'll assume these fit in a machine-word (the same assumption for van Emde Boas queues) — so a machine-word has $\log m$ bits (to hold these numbers). We'll go beyond indirect addressing and assume we have constant-time math operations — we can add and multiply these integers in constant time. (This is not actually true — if you look at the circuitry multiplication is not constant — but we'll assume it.)

We want to do lookups for equality — is this key in the table or not? (Usually you'll store values associated to the keys, and want to look them up.)

An array is a perfectly good solution — make an array with indices $1, \ldots, m$, and put the value associated to key k in the kth box. (You might worry about how we initialize this array, but there's a trick for using arrays without actually initializing them — you just have to triple the space you use.)

But this takes up too much space. The idea of hashing is to make a smaller array, and to store the key k in a bucket $h(k)$ for some *hash function* $h\colon \{1, \ldots, m\} \to \{1, \ldots, s\}$ where s is the size of the array you want to allocate.

§6.3.2 The problem

This seems great — to store an item, I just compute its hash function and put it there; to check if I have the item, I compute $h(k)$ and look at that bucket. But what can go wrong?

The (only) problem with hashing is *collisions* — when two items are inserted in the same bucket, meaning $h(k_1) = h(k_2)$. This is not really a problem; you can fix this by putting a linked list at the bucket. Now however many items hash to that bucket, we just string them out into a linked list, and the collision problem has gone away.

But this creates a new problem. The new problem is that if there's too many collisions, then you could have a really long linked list. In the worst case, all items might end up in the same bucket, in which case the hash table has no benefits (we still have to scan through the entire linked list to find whether the element is present).

So the challenge is to figure out a hash function with few collisions.

In a certain sense, this is impossible. We cannot develop a hash function that hashes $[m]$ to $[s]$ with no collisions, because of the pigeonhole principle; and if $m \gg s$, then there'll be a tremendous amount of items that *must* all hash to the same bucket. So it seems we're doomed — there's no good hash function. But we want a good hash function.

§6.3.3 Hash families

So the problem is that there's no universally good hash function. The fix is to be *instance dependent*. What we do is we define a *hash family* — a *set* of hash functions, such that at least one is good for any input set of items that we want to hash. For now we'll focus on the static case — we're given a collection of items, and want to find a hash function to put those items into a dictionary with fast lookups (we won't worry about inserting and deleting functions as we go).

What features do we want? One hash family we could use is that if we're trying to put s items into an array of size s, we could sort all the items and put the kth item into the kth position of the array. This is *a* hash function — there is a function that does this, and it gives us constant-time lookups in a sense. But in another sense, it's useless. The point is that *computing* that hash function would take a long time. There's the problem of figuring out which function to use (which involves sorting), and also evaluating the hash function when given a key (which here involves doing a binary search).

So what we need is a *small set of hash functions that can be evaluated quickly*. Why do we say *small set*? If you have a gigantic number of functions in your set, then just writing down a description of which function you're using will take lots of space. And to evaluate the function you need to know which function you're using, so you have to read that description, and that takes a lot of time. This means you want your set of hash functions to be small (and quickly evaluating).

So what might we do? This is where David gives us a preview of randomized algorithms, because the answer is to randomize.

§6.3.4 Random hash functions

As a detour, let's imagine what happens if we do a *random* assignment of items to buckets (i.e., we use a random hash function). We could do this; the problem is that it would take $\Omega(m)$ space to store. But let's ignore this for now; what kind of collision behaviour would we get?

Imagine we have a set of n items, and we map them *randomly* to a size-s array. What can we asy about the collisions that will happen? If $n = s$, this is the same as the 'balls and bins' problem, and there are strong theorems saying that the maximum number of items in a bin (we're going to use 'bin' and 'bucket' interchangeably) is

$$O\left(\frac{\log n}{\log\log n}\right)$$

with ver yhigh probability. With randomization nothing is ever certain (there's a chance all the items end up in the same bit), but that's incredibly unlikely.

This is already better than binary search trees — you only have to look at $\frac{\log n}{\log\log n}$ items. But you can do better — what really matters is when we do a lookup, how many items do I run into? And even if there's one big bin, it's unlikely I'm in that big bin. So we actually need to look at the expected behavior for an arbitrary item that gets thrown into the hash table — how many items is it going to collide with?

Randomized algorithms uses about four concepts from probability, and all the rest is algorithms; we're going to see all four here. How do we analyze the expected number of collisions a particular item will see? We'll analyze this using the tool of *indicator variables*; we define

$$C_{ij} = \begin{cases} 1 & \text{the } i\text{th and } j\text{th items land in the same bucket} \\ 0 & \text{otherwise,} \end{cases}$$

so C_{ij} is a random variable. Then C_{ij} is 1 with probability $\frac{1}{s}$ and 0 otherwise.

Now we observe that the total number of collisions between item i and any other item is just $\sum_{j \neq i} C_{ij}$. And we are interested in the *expectation* of this value (this is the second critical concept). We evaluate this using *linearity of expectation* (this is the third important concept) — we have

$$\mathbb{E} \sum C_{ij} = \sum \mathbb{E}[C_{ij}] = \frac{n-1}{s}.$$

So that's the expected number of collisions with item i, or equivalently the expected amount of time to look up item i (because collisions are the only thing that can distract you). In particular, if $s \geq n$, then this is at most 1. So as long as the hash table is as big as it has to be, you *expect* just one collision.

§6.3.5 Carter–Wegman universal hashing

So it seems a random function is great; the downside is that you need linear space in the *universe size* to write down its description. This is where we get to universal hashing.

The key insight is we don't need an *entirely random* function. Our analysis only worried about *pairs* of items colliding. So we only need a function that's random *enough* to have the right behavior on *pairs* of items. We don't need *full* independence on arbitrary subsets of items, we just need the right expectations to come out of those *pairwise* collision variables. In other words, it's sufficient to have just *pairwise independence* — all that means is that the value of $h(x)$ is uniform conditioned on the value of any *one* other $h(y)$.

This happens automatically if we have full independence (if you have a truly random function, $h(y)$ tells you nothing about $h(x)$). But with full independence, you also have that knowing $h(y)$ and $h(z)$ tells you nothing about $h(x)$. We don't need that, just pairwise independence.

So Carter–Wegman had the idea of choosing a function that maps items to buckets in a pairwise independent fashion. For this, we'll take a tiny detour into algebra or number theory. We pick a prime number p in the range $\{m, \dots, 2m\}$. Now we pick two numbers $a, b \in \mathbb{Z}_p$, and define a function

$$h_{ab}(x) = (ax + b \pmod p) \pmod s.$$

(We take this mod s in order to fit all of this into the array, since p is still pretty big.)

> **Claim 6.3 —** If we choose $a, b \in \mathbb{Z}_p$ at random, then $ax + b \bmod p$ and $ay + b \bmod p$ are uniform and pairwise independent.

So by choosing two a and b, we're choosing a hash function which randomly distributes our items in the range $\{1, \dots, p\}$. Then taking them all mod s doesn't really mess up the randomness; they're still pretty much uniformly distributed. So this is enough to give us the appropriate probabilities for collisions of pairs of items — since these are pairwise independent, knowing the bucket for x doesn't tell you anything about y, and so the analysis of the expected number of collisions remains true. So that's why we get a good hash function this way.

Now we'll prove pairwise independence.

Proof. We need to show that $ax+b \bmod p$ and $ay+b \bmod p$ are independent (and uniform), when we choose a and b at random. Fix some pair $x \neq y$. Suppose we consider the probability of getting output values

$$ax + b = s \quad \text{and} \quad ay + b = t.$$

To prove pairwise independence, we want s and t to be uniform and independent; this means for any particular values of s and t, we should have

$$\mathbb{P}[(ax + b = s) \text{ and } (ay + b = t)] = \frac{1}{p^2}.$$

Now we can write this in matrix notation — this is

$$\mathbb{P}\left[\begin{bmatrix} x & 1 \\ y & 1 \end{bmatrix}\begin{bmatrix} a \\ b \end{bmatrix}\right] = \begin{bmatrix} s \\ t \end{bmatrix}.$$

And we can invert this matrix — this is just

$$\mathbb{P}\left[\begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} x & 1 \\ y & 1 \end{bmatrix}^{-1}\begin{bmatrix} s \\ t \end{bmatrix}\right].$$

And this is just $\frac{1}{p^2}$, since we're looking at the probability that a and b take on particular values (and they're independent).

Can we do this inversion? It's important that $x \neq y$, so that this matrix is invertible. It's also important that p is prime (division becomes weird otherwise, because you don't have a field; but if you do have a field then inverses are well-defined, so we get an actual number on the right-hand side). □

Student Question. *Why did we require $p \leq 2m$?*

Answer. The point is just that we don't want to use huge primes because that'd be inefficient.

This basically completes the story of universal hashing. When we pick a and b we store them as the representation of our hash function. So our hash function takes just two machine-words, and it's computable in constant time. So it meets the ideals of what we want — it introduces enough randomness that the expected number of collisions is constant, giving the performance we want.

But looking ahead, it's expecting a constant number of collisions, but what if you want *no* collisions at all? If you have n items, you *can* put them in a size-n array with no collisions. Next class we'll see how to generalize this idea to get *perfect* hash functions with no collisions at all (for n given items), making them as good as an array.

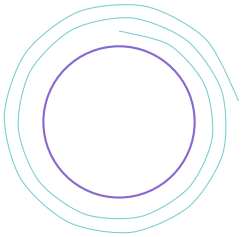
§7 September 18, 2024

§7.1 A pairwise independent hash family

We'll pick up where we left off last lecture; we saw we can use the $ax+b \pmod p$ construction to get pairwise independent hash functions. That already gives us $O(1)$-size lookups in a *prime*-sized table. But what if the size s of our table is *not* prime? We've actually already done all the hard work. We're going to choose $p \gg s$ (we'll come back to how big it should be in a minute) and now we slightly modify our definition — as before, we compute $ax + b \bmod p$, but now we take that whole thing and compute it mod s. So

$$h_{ab}(x) = ((ax + b) \pmod p) \pmod s.$$

We claim this is just fine — we're still going to have a basically uniformly distributed hash table with $O(1)$ lookups. Why? The point is that we know $ax + b \pmod p$ spreads things out uniformly and pairwise independently; and so then when we collapse that to a table of size s, things will be *close* to uniform, and still pairwise independent. Pictorially, you can imagine thinking of the table of size s as a ring; and now our table of size p can be thought of as wrapping around and around this ring. And any item that gets computed mod p then gets projected down to the corresponding bucket in the table of size s.



From this picture, we can see every bucket has about the same probability of getting an item — items are uniformly placed on the spiral, and every bucket has about the same number of origins on the spiral. They won't have *exactly* the same number because $s \not| p$ (p is a prime); but we're only off by 1. So each bucket has the same number of spiral elements up to ± 1. And that means the probabilities for the different buckets only differ by something like $\pm\frac{s}{p}$. So for example, if we set $p \approx n \cdot s$, then the bucket probabilities differ by only $\frac{1}{n}$; and looking back at the analysis, this only changes our expected number of collisions by $O(1)$.

It's crucial that in addition to being significantly larger than x, we also need p to be bigger than the maximum key size. This is because if you had two keys that differed by p, then they'd always map to the same location, and our argument for pairwise independence breaks down. (In that argument we did everything mod p, so we were assuming $x \neq y \pmod{p}$.)

But p is just an integer — not a data structure. So it can be really big; it can even be 2^{64}, and we're fine. (That's not prime, but lots of things work even with prime powers.)

§7.2 Conclusions

We've shown that by choosing a random hash function from this *two-universal hash family*, we get pairwise independence for the position of items, and this means we have an *expected* load of a constant. So we have $O(1)$-time lookup and delete, as long as our table size is larger than n.

Here all we need to do is store two integers a and b, and we have $O(1)$ operations in expectation. Expectation is nice in general — over a large number of operations, you expect things to regress to the mean, so your average cost will almost certainly converge to the expected cost. But it's interesting to ask about the maximum load — how many items can end up in the same bucket with some probability? It turns out you can prove that the maximum load is $O(\sqrt{n})$ with probability $1 - \frac{1}{n}$. So if you use this pairwise independent hash function, it's very unlikely you'll have more than $\sqrt{n}$ items in a single bucket. (This bound is when $s = n$.)

More generally, it's quite likely that *some* items will have a larger load (the $O(1)$ is only an expectation). But fortunately, they're sort of 'random' items, and they're unlikely to be searched for often. So the fact that some of your operations are a bit slow doesn't affect the overall expected runtime too much.

> **Remark 7.1.** As an interesting sidenote, theoreticians have proven that there do exist pairwise independent hash families where the $\sqrt{n}$ maximum load is very likely. But it only involves $\sqrt{n}$ items, so it doesn't disrupt the average; and you still get an average lookup cost of $O(1)$.

So we're now done with getting a data structure that's good in expectation. More generally, we've actually shown an expected cost of $1 + n/s$, where s is our space and n our number of items. In particular, when we set $s = n$, this becomes $O(1)$ — so in linear space, we can store items with expected $O(1)$ operation cost. (Obviously if the space is much less than n then we can't do that; the point is we don't need more than n.)

> **Student Question.** *What if you access the same element every time?*
>
> **Answer.** Yes, this could give you bad worst-case runtime; our guarantees are just in expectation.

§7.3 Motivation for perfect hashing

> **Question 7.2.** Can we *guarantee* $O(1)$ lookups?

We said that if you pick a random hash function, there's a good *chance* that it's good, but there's always a problem. So how do we get around that? We assume we're *given* our keys in advance. It's true that for a random hash function, there's always a bad set of keys. But if I know what the keys are, I won't use that hash function; the question is, can I always find a good hash function if I'm allowed to look at the keys when hunting for the hash function?

In fact, we'll be really ambitious; we'll try to arrange for *no* collisions at all. This will be what's called a *perfect* hash function.

Obviously, if we want a perfect hash function, we'll need $s \geq n$. But nothing we've done yet lets us get a perfect hash function for *any* space at all.

§7.4 A starting point — perfect hashing with quadratic space

Let's creep up on this notion of a guarantee.

> **Question 7.3.** What's the *chance* of no collisions if we use a random hash function?

We'll creep up on this in a standard way — we'll consider the *expected* total number of collisions. Using the same indicator variables as last time (where C_{ij} is the indicator of the event that i and j collide), we have

$$\mathbb{E}[\text{total collisions}] = \mathbb{E}\left[\sum C_{ij}\right] = \sum \mathbb{E}[C_{ij}] = \sum \mathbb{P}[i,\ j \text{ collide}]$$

(note that here we have a *double sum*, unlike last time, since we're looking at all collisions rather than collisions with a fixed element). And the probability that i and j collide is $\frac{1}{s}$, while the number of terms in the summation is $\binom{n}{2}$ (we're looking at all pairs of items); this means

$$\mathbb{E}[\text{total collisions}] = \binom{n}{2} \cdot \frac{1}{s} \sim \frac{n^2}{2s}.$$

In particular, if $s \geq n^2$, then this quantity is at most $\frac{1}{2}$. So if we use quadratic space, the expected number of collisions will be $\frac{1}{2}$.

That's pretty nice, but $\frac{1}{2}$ is not 0. On the other hand, you can't really have half a collision. So we can use a well-known method to get from half a collision to *zero* collisions. Last time we mentioned four things from probability — indicator variables, linearity of expectation, independence, and pairwise independence; now we're going to use what's called *Markov's inequality*, which is the last thing you need to know.

> **Theorem 7.4** (Markov's inequality)
>
> If X is a nonnegative random variable, then for all $t > 0$ we have
>
> $$\mathbb{P}[X \geq t] \leq \frac{\mathbb{E}[X]}{t}.$$

This is just a formalization of the rule that 'not everyone can be above average' — if there's a large probability that $X > t$, then all that probability is going to contribute to the expectation of X and make it too big.

In particular, in our case, we have

$$\mathbb{P}[\#\text{collisions} \geq 1] \leq \frac{1}{2}.$$

So if we pick a random hash function into quadratic space, then there's at least a 50% chance that there will be no collisions.

This is the most common way to move from an expectation to a probability, using Markov's inequality. So now we have a probability of getting a perfect hash function. But we still don't have certainty. How can we turn this probability into certainty?

We can start by trying to generate one hash function — we try a random function, and if it's perfect (i.e., there are no collisions), we can stop and we're done. Note that we have a 50% chance of success. But what should we do if we fail? We just try again — we loop back to the beginning and try another random hash function.

Now this algorithm is guaranteed to succeed, because it keeps trying until it succeeds. So it's guaranteed to *eventually* find a perfect hash function. How long is it going to take? Well, there's two things to ask about — how many iterations of the loop are there, and how fast can we run each?

First, how fast can we run each iteration? Generating a random function is easy (we're still working with pairwise independence here, and we know it's easy to generate a random pairwise independent function). How quickly can we check if the resulting function is a perfect hash function? We can just fill the hash table and see if there are any collisions, so that takes $O(n)$ time.

What can we say about the *number* of iterations? Well, this is random, so it doesn't have a specific value. But what can we say about the *expected* number of iterations? We have

$$\mathbb{E}[\#\text{iterations}] = \frac{1}{\mathbb{P}[\text{success}]} \leq 2,$$

since each trial has probability $\frac{1}{2}$ of succeeding. (In general, if you roll a k-sided die and stop when you see 1, the expected number of rolls is k; this intuitively makes sense because if you roll a die k times then you'd expect to see every number once.)

So now we have a mechanism for generating a perfect hash function. In fact, we have *two* mechanisms. The algorithm we started with, with a 50% chance of success, is a *Monte Carlo* algorithm — it's always fast and often works. What we defined by doing iteration is a *Las Vegas* algorithm, which means it always works and is often fast. You can basically turn any Monte Carlo algorithm into a Las Vegas algorithm by repeating until it succeeds. (You can also go the other way, but it's more complicated to describe.)

§7.5 Decreasing space — a first attempt

So now we have algorithms for perfect hashing, but we're using $O(n^2)$ space for n items; that doesn't seem efficient.

> **Question 7.5.** Can we use less space?

Maybe the bound of $\frac{n^2}{2s}$ we had was not tight; might it be possible to prove a better bound and say we have a good chance of being perfect, even with a smaller table? Well, here's a concrete example showing the answer is no. We have some number of students in the class; suppose David asks all the students to state their birthdays. So David is hashing all of us according to our birthdays; we have 365 buckets and 37 students. Is the chance of a collision small?

If it were $\frac{1}{2}$, we'd be happy. But it's actually very large. This is an instance of the *birthday paradox* — once you have more than 23 people, the probability of a collision is greater than $\frac{1}{2}$, and by the time you get to 37 it's very large.

In general, $\frac{n^2}{2s}$ is pretty tight. The reason for this is, imagine we put the people into buckets one at a time. The first person has no collisions. For the second person, the probability they don't collide is $1 - \frac{1}{s}$. And

then the next person is $1 - \frac{2}{s}$, because there's two occupied buckets. And so on; so

$$\mathbb{P}[\text{no collisions}] = \left(1 - \frac{1}{s}\right)\left(1 - \frac{2}{s}\right)\cdots\left(1 - \frac{n-1}{s}\right).$$

Using the approximation $1 - x \approx e^{-x}$ for small x, we get that this is roughly

$$e^{-(1/s+2/s+\cdots+(n-1)/s)} \approx e^{-n^2/2s}.$$

This is the probability of having no collisions; so once n^2 is bigger than s, the probability of having no collisions becomes very small. This means if we don't make our hash table quadratic in size, we will almost certainly see collisions.

§7.6 Perfect hashing — the solution

How can we get past this? We can use a very clever trick used to Fredman, Komlos, and Szemerédi (Fredman was from Fibonacci heaps; Komlos gave very interesting results on spanning trees; Szemerédi is a combinatorialist). They suggested addressing this by introducing a *second level* of hashing.

What you first do is you hash your items to $O(n)$ space. This is going to give you *some* collisions, but not too many. And that means you need a lookup data structure in each bucket. We were previously putting a linked list there. But why use a linked list? Let's instead build a perfect hash table *on each bucket*! So if we have b items in the bucket, then we use b^2 space in order to make it perfect.

This gives us $O(1)$ lookups — when we're given a key, we compute the top-level hash function to get a bucket, and then inside that bucket, we use the perfect hash function for that bucket to find the item in $O(1)$ time. So the only question is, what's the total space? Well, the top level takes $O(n)$ space (where we store the information about the buckets). Then we have to add up, over all buckets, the square of that bucket size — so we want to consider

$$n + \sum_k b_k^2,$$

where b_k is the bucket size. Now this is where the authors do a very clever trick — what is b_k^2? That's the square of the bucket size, but we can also think of the bucket size as a sum of indicator variables, where we look at all the items and have an indicator for which of them ended up in bucket k. So we can write this as

$$\sum_k b_k^2 = \sum_k \left(\sum_i 1[i \text{ lands in bucket } k]\right).$$

Now we've got a sum squared, and we can expand that out as

$$\sum_k \sum_{i,j} 1[i \text{ lands in bucket } k] \cdot 1[j \text{ lands in bucket } k].$$

Now we're taking a sum of products of pairs of indicator variables. And this product is telling us whether or not there's a collision in bucket k — specifically, this is

$$\sum_k \sum_{i,j} 1[i \text{ and } j \text{ collide in bucket } k].$$

And if we sum over all buckets, we're computing the total number of collisions — i.e., this is just $\sum_{i,j} C_{ij}$.

And now we can put an expectation around the whole analysis; and we get

$$\mathbb{E}[\#\text{collisions}] = \mathbb{E}\left[\sum C_{ij}\right].$$

And we've calculated this before! We do have to be a bit careful, though, because here we're summing over *all* pairs (i, j), including ones where $i = j$. (Before we only looked at pairs $i < j$.) So we actually have to separate out the terms with $i = j$; then we can write this as

$$\mathbb{E}\left[\sum_i C_{ii}\right] + 2\mathbb{E}\left[\sum_{i<j} C_{ij}\right].$$

The second term we've seen already; it's $\frac{n^2}{2s}$. And the first term is n (every item collides with itself).

So if we set $s = O(n)$, then we get

$$\mathbb{E}\left[\sum b_k^2\right] = O(n).$$

In other words, if we use a size-n top-level table, and then use quadratic second-level tables, then the expected space usage is $O(n)$. So we get linear space and constant-time lookups.

Finally, how do we get from *expected* space usage to a *guarantee*? We use the same Markov approach — if the expected space is cn, then what we do is we generate one of these two-level tables, and if the space used is less than $2cn$, we stop. Otherwise, we repeat.

Why does this work well? We're using the same paradigm as before — Markov's inequality says that our success probability is greater than $\frac{1}{2}$, which means the expected number of times we need to try this is 2. So we get a Las Vegas algorithm for generating a perfect hash table, with no collisions whatsoever.

> **Remark 7.6.** To wrap up, let's talk about this constant c. If you work through this argument, you get $c \approx 13$ — you need $13n$ memory locations to get a perfect hash function in this way. With a bit more work, you can get this down to 6. In the Fredman–Komlos–Szemerédi paper, they actually get it down to $1 + o(1)$, which is kind of incredible — the amount of space you need to use for $O(1)$ lookups is the same as the amount of space you'd need using just an array you scan through every time.
>
> For this, they do some clever tricks. First, when you build your top-level hash function, some items are going to end up with no items at all — in fact, a constant fraction will be empty. And they use those empty buckets to hold the extra information you need for the second-level hash tables. (This is very contorted and you'd never do it in practice. But it shows perfect hash functions are really perfect.)

§7.7 Dynamic perfect hash tables

The method we described for building perfect hash tables required checking that it's perfect by hashing everything. This requires that you already have all the items. But what if we want to support insert and delete? (Then someone might insert items that destroy perfectness.)

Figuring this out took a couple more years. The trick is that any time your hash table becomes imperfect, you just build a new one. You prove you don't have to do this too often, and you end up with an $O(1)$-time amortized data structure.

§7.8 Intro to combinatorial optimization

That's the end of our data structures unit, and we'll now switch to talking about combinatorial optimization. So far, we've been doing data structures, where you store things and look things up. Now we'll think about having an input, and wanting to compute an optimal output for it. So now we won't be talking about runtime per operation; we'll just talk about the runtime of an algorithm.

Generally, any combinatorial optimization problem has a set of *feasible solutions* — those which are allowed. And each of those feasible solutions has a *cost* or a *value*. And you want to find the feasible solution with optimum cost or value. (Sometimes you're trying to minimize cost; sometimes you're trying to maximize value.)

§7.8.1 A paradigm

For many of these problems, there's a pretty common paradigm for how you think about them. First, you understand feasibility — you ask, what defines a feasible solution? You may talk about actually having an algorithm to *determine* feasibility. Sometimes even *that* is hard.

Your next step might be an algorithm to find *any* feasible solution.

And after you know how to find feasible solutions, you start thinking about solution values. So your first step might be to develop an algorithm to verify the optimality of a given feasible solution. And only when you've gotten that far do you think about how to compute an optimal solution.

Generally it's pretty daunting to see a new problem and immediately think about computing an optimal solution; but you can get there more easily through these steps.

§7.9 The maximum flow problem

Our first problem will be maximum flow. All we'll do today is define it.

In Max-Flow, we're given:

- a directed graph G with m edges and n vertices;
- a specified *source* vertex s and *sink* vertex t;
- a *capacity* u_e for each edge e.

A *flow* is an assignment of a number f_e to each edge e. We say a flow is *feasible* if it satisfies:

- The *capacity* constraints: $0 \leq f_e \leq u_e$ for all e.
- *Conservation* constraints: for every v, we have $\sum_w f(v,w) - \sum_w f(w,v) = 0$, except when $v = s$ or t. (In words, this says the total amount of flow entering a vertex equals the amount of flow leaving the vertex.)

When we have a flow, we say its *value* is $\sum_v f(s,v) - \sum_v f(v,s)$ (this is of the same form as in the conservation constraint, but it's not 0). And the goal is to compute the maximum value of a flow.

> **Remark 7.7.** This models many problems; but the intuitive one is you have a collection of pipes, and a source of water at some place; and you want to know how much water you can push to some other place in this network of pipes. There the capacities and flow values will be given in some rates, like gallons per second; but they're just numbers and you can add them up and subtract them the way we're doing here.

This is the central problem in combinatorial optimization. It was first written about by Ford and Fulkerson in 1956. There's two reasons for this. One is that there's all sorts of interesting theory you can write about this problem; it's complex, but you can prove all sorts of interesting facts and algorithms. The other reason it's central is that tons of real-world problems can be converted into flow problems. Sometimes the conversion is direct (e.g., moving traffic in a city), but sometimes it takes more creativity.

And over the decades, we've developed incredibly fast flow algorithms — in practice you can get close to linear, though theory is a bit behind practice. We'll spend the next week unpacking this problem and developing algorithms for it, and getting a sense of how one does combinatorial optimization.

§8 September 23, 2024

§8.1 History of max-flow

This week is going to be about max-flow. Max-flow is the canonical problem in combinatorial optimization. We're going to spend several weeks on combinatorial optimization (before moving to other models). It's

about optimizing things combinatorially — you have some objective function to maximize or minimize, and your choices involve certain combinations of things. They also involve numbers of various sorts, but the solution is often defined by a subset, even if you also have to put numbers onto the subset to make it clear what the solution is.

If you've seen max-flow before, in this class we'll hopefully go deeper. If you haven't seen it, it's a really important problem — one reason is it's algorithmically rich. It's also broadly applicable. There are many problems in the real world that when you look at them properly can be formulated as max-flow problems or variants. David has had this experience in his own research twice — someone from some other area gives him a very interesting research problem, and David says 'oh, that's a max-flow problem!' and gets his name on a paper.

And there are also great algorithms for max-flow — the theory is powerful, the time-bounds you can prove are great, and the performance of algorithms in practice is spectacular (you can get close to linear time in practice).

The problem was first studied during WW2 and afterwards. In Russia, they were trying to optimize for how to use their rail networks to ship goods around. In the US, they were trying to figure out how to use the fewest bombs to destroy Russia's network so they could not ship commodities around. It turns out these are actually the same problem, as we will see.

The earliest algorithms were general-purpose (something called *simplex*). But Ford and Fulkerson, in seminal work from 1956 and 1961, initiated the study of specialized algorithms for max-flow. That has continued even to today; people are still making progress on max-flow. However, the most recent progress uses continuous optimization, which David dislikes; so we won't see that, but we'll see many discrete algorithmic insights that have powered the field for a long time.

§8.2 Definition of flows

Definition 8.1. Suppose we have a graph G with a *source* vertex s, a *sink* vertex t, and nonnegative capacities $u(e)$ associated to edges e.

A flow is an assignment of a flow value $g(e)$ to each edge e, satisfying certain constraints:

- *Conservation*: for every vertex $v \neq s, t$, we have $\sum_w g(u, w) - g(w, v) = 0$.
- *Capacity*: $0 \leq g(e) \leq u(e)$.

We define the *value* of the flow as $\sum_w g(s, w) - g(w, s)$.

One deep mystery David will solve weeks from now is, why do we use u for capacities? That will become clear next week or the week after, but this is the standard for flow problems. Why nonnegative vs. positive? A zero capacity is essentially the same as not having an edge at all; so you can say positive capacities if you want to ignore the zero-capacity edges, and nonnegative if you want to keep them around and not use them.

In words, conservation says the amount of flow going out of a vertex is the same as the amount of flow going into a vertex, unless the vertex is the source or sink. The value is measuring the net amount of flow that comes out of the source s. You might wonder about t, but it turns out all the flow has to go somewhere, and the only place it can go is t; so the amount of flow going into t is the same as the amount of flow going out of s, and we could use either definition (though we'll have to prove this).

This is the *gross flow* definition, but sometimes it's more convenient to work with a *net flow* definition. This means instead of looking at flow from one vertex to another, we look at a pair and combine the flow in both directions, to get a *net* flow from one to the other.

Definition 8.2. Suppose we have a graph G with a *source* vertex s, a *sink* vertex t, and nonnegative capacities $u(e)$ associated to edges e.

A net flow is an assignment of a flow value $f(e)$ to each edge e, satisfying certain constraints:

- *Skew symmetry*: $f(v,w) = -f(w,v)$.
- *Conservation*: for every vertex $v \neq s,t$, we have $\sum_w f(v,w) = 0$.
- *Capacity*: $f(e) \leq u(e)$.

We define the *value* of the flow as $\sum_w f(s,w)$.

Note that now we're allowed to have negative flow values (if we're measuring against the direction of flow). But because capacities are nonnegative, the constraint in one of the two directions will impose a restriction on the net flow.

The net flow is cleaner mathematically if you want to think about linear functions and combining them (where you want to negate and subtract). But gross flow is much better for intuition about what's going on in a flow problem. So we'll often talk in the gross flow setting. If you use modern linear algebraic techniques, then you're always talking about net flows, because you're doing linear algebra.

The intuition is you have a big hose hooked up to s, you're pumping water into it, and it flows through pipes (the edges of the graph). The only drain is at t; so water flows through the graph, and eventually exits at t. And you're interested in how much water you can pump through the network. Here you'll be measuring volume per unit time (e.g., gallons per second); but of course for our calculations that's just a number.

Problem 8.3 (Max-Flow)

Find a feasible flow (i.e., a flow satisfying the above constraints) of maximum value.

Technically, you can very generally talk about a flow as an assignment of numbers to the edges; but we're only interested in feasible flows, ones satifying these constraints.

§8.3 A recipe for combinatorial optimization

Generally, you don't want to jump straight to 'how do I solve this optimization problem?' Instead, you creep up to it in smaller steps.

(1) Understand what *feasible* solutions look like. What can we infer just from the fact that a flow satisfies these constraints? How can we look at a flow and decide if it satisfies these constraints? What other consequences are there?

(2) Understand what characterizes an *optimum*.

(3) How do you verify that a candidate solution *is* optimum? This is far easier in some cases than finding an optimum solution — here someone gives it to us, and we verify that it is indeed optimal. This is a principle David has been thinking about lately, the idea of separating verification from optimization. This is at the core of NP-completeness — a whole class of problems where it's hard to compute a solution but easy to verify. With LLMs, things may become all about verification — the LLM gives you an answer but you can't trust it, and there'll be lots of problems about taking an answer given to you and deciding if it is correct.

After we understand how to verify a solution, we often understand how to detect *non*-optimum solutions. And if you can explain what makes a solution non-optimum, that gives you a hint for how to *improve* the solution.

(4) After we understand that much, then we can talk about actually finding the optimum.

So that's what we're going to do today.

§8.4 Understanding flows

First we'll get some deeper understanding of what flows look like. Thinking about flows as sending water through a network, what's the simplest example of a flow? It's a path — a path is a kind of trivial example of a flow (you start at the source, send some amount of flow to vertex 1, send the same amount of flow out of vertex 1 to vertex 2, and so on; eventually it reaches the sink t).

Slightly harder, what's an example of a flow that's still simple but is not a path? You can also have a loop — that also satisfies the constraints (at least, it satisfies the conservation constraint; it satisfies the capacity constraints if the amounts are small enough).

It turns out these are actually the *only* interesting flows, and all other flows can be written as combinations of them.

Lemma 8.4 (Path decomposition lemma)

Any flow satisfying conservation can be decomposed as a sum of at most m flow paths and flow cycles.

Of course, on each path and cycle, we also need an amount; but structurally it's just paths and cycles. This is very useful for intuition about flows — it means you really just need to think about paths and cycles and what happens to them.

Proof. Well, we're going to construct a decomposition. Given a flow, we're going to try to find a path or a cycle, and take it out of the flow in a way that simplifies the flow (and then we can iterate).

We're actually going to do this using induction on the number of edges with nonzero flow.

Given a flow, how do I know there is some path or cycle? Let's just hunt at one. We start at s. If the flow is nonzero, then there has to be an edge going out of s with positive flow (by the definition of value). Suppose this edge goes to v_1.

Then there is flow entering v_1, which implies there must also be flow *leaving* v_1 to some v_2.

We continue until something happens that lets us stop. What can happen that tells us we got a step we can use to simplify our flow?

Case 1 (We reach the sink). If we reach the sink, then we've just traced out a path in the graph. Now, how can we take that path out in a way that simplifies the graph? Some edge on this path has the minimum amount of flow; if we assign that value to all the edges on that path, then it's a path carrying that amount of flow which uses up all the flow on the min-flow edge. This reduces teh number of edges carrying nonzero flow by at least one (at least one new edge now has zero flow).

Case 2 (We revisit a vertex). Then this means we've found a cycle of flow. (There's a path at the start, so it looks sort of like a lollipop; but we can ignore the path that took us to the cycle, and there is a cycle of flow.) And then we can remove this cycle in the same way.

So either we find a path or a cycle (we have to stop eventually, because there's only n vertices and each time we take a step we use one up). So we're done; we've shown how to take any flow and decompose it into paths and cycles.

In fact, we can actually say more — the number of paths and cycles involved in this decomposition is at most m (the number of edges), because each step gets rid of at least one edge. □

Remark 8.5. We left out a detail here — once we're done, we have nothing coming out of s, but there might still be some flow. If there's still flow, then we can start at a vertex that has exiting flow. Then when we do this procedure, we're going to trace out a path of vertices; and each time we get to a vertex, because we have incoming flow, we will also find an edge with outgoing flow. So again, if we keep this up, we will eventually hit a vertex we've seen before (maybe the one we've started at, or maybe another); and we can pull out that cycle in the same way.

With flow decomposition, we can already start to understand more about the decision questions (before we get into optimization).

Question 8.6. Is there a positive feasible flow — more specifically, how can we look at a graph and decide if there is a positive flow?

Claim 8.7 — There is a positive feasible flow if and only if there is a connection from s to t (using nonzero-capacity edges).

Proof. Suppose there is a nonzero feasible flow. Then the path decomposition that we just proved shows that you can divide this into paths (from s to t) and cycles. The value of the flow is linear, so you can add up what's going on with the paths and cycles. A flow cycle contributes 0 to the value of the flow (this is true even if the cycle goes through s, because it puts as much flow into s as it takes out). So only the flow paths contribute to the value of the flow. And so if you look at this decomposition, the cycles aren't helping the flow be positive; this means there must be some path(s) of nonnegative capacity.

Now let's look at the other direction — suppose there is *no* feasible flow. Can you say there is not a path? Yes. We can show this by contrapositive — if there was a path from s to t (of positive capacities), then you could use that path to send a positive flow. □

But we'll also give a direct proof, because it's useful for defining something. Let's consider the vertices reachable from s with positive capacities; let's call the set of such vertices S. Note that $t \notin S$ by assumption.

We claim that S defines a *cut* with 0 capacity leaving it. In other words, we've got s and t, and we can find a collection of vertices reachable from s on nonzero capacities. Then there are other vertices in the graph, including t; but there's no edge of positive capacity from the s-side of the cut to the t-side (i.e., everything not on the s-side — this side might have some stuff not at all connected to t).

This is what is known as an *s-t cut*, and the *value* of such a cut is defined as the sum of capacities exiting the cut.

If there is no feasible flow of value greater than 0, then this procedure gives us a *cut* of value 0 separating s from t.

§8.5 Verification of max-flows

Question 8.8. How do you decide if a flow is optimum?

If you want to show that something is maximal, you need some sort of upper bound — you have a flow and it has a certain value, and you want to prove no flow has larger value, which means you want to prove an upper bound on the value of the optimum.

We claim that these cuts we've just constructed provide such upper bounds. Consider any flow f, and some s-t cut (S, T). And let's consider what happens if we decompose the flow into paths and cycles. Every path must cross from S to T at least once. (This is hopefully obvious, but the continuous version is not obvious — they had problems in the 19th century until they proved the Jordan curve theorem. It can be proved by induction.)

And if the path is exiting, then it uses up some capacity of an edge across the cut. And the paths can't share this capacity. So that implies the outgoing total capacity of the cut has to be at least the value of the flow (cycles don't contribute to the value of the flow; the value of the flow is just the sum of *path* amounts, and every bit of path crosses every cut). So what we've just proven is the following:

Claim 8.9 — We have min-cut capacity $\geq$ max-flow.

So we have an upper bound. Now that we're talking about cuts, we'll also mention a corollary of the path decomposition lemma.

Lemma 8.10

The net flow crossing an s-t cut is equal to the value of the flow.

Why? Again we rely on path decomposition. A flow is made of paths and cycles, and value comes from the paths. How much does a path contribute to the net flow? A path can go in and out of the cut may times; when it goes out it adds to the net flow, and when it goes in it subtracts. But overall, a path has to leave exactly once more than it enters the cut (if it's a path from s to t); so what it contributes to the flow across the cut is exactly the value of that path.

On the other hand, a cycle crosses the same number of times in both directions, so it contributes nothing to the net flow.

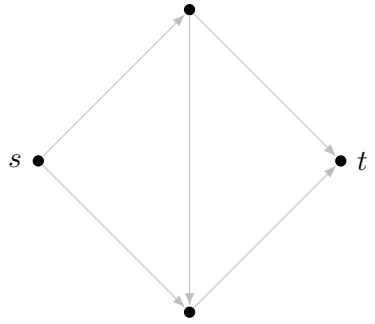
§8.5.1 Augmenting paths

So we've now proven an interesting upper bound on the max-flow. Now we've got a mechanism for deciding whether there's a nonzero flow or not; we'll now think about the problem of *verifying* whether a flow is maximum (equivalently, deciding whether a flow is *not* maximum and we can send more flow).

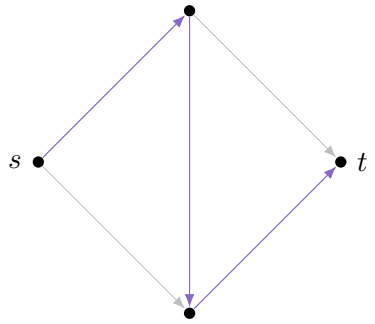
Question 8.11. How do we verify that a flow is *not* maximum (i.e., can be improved)?

The flow that we have is using up some of the capacity in the graph, so we can look at the capacity that remains. In what case would it be *obvious* that we can send more flow? Well, if there's a path that still has capacity, then you can use that path to send more flow.

We might hope that's an if-and-only-if. But that's not true; here's an example.



Assume all edges have capacity 1, and our flow is simply the following path.



There is not more capacity left over that we could use to send more flow. But this is not a max-flow — we could have sent one unit along the top and another along the bottom.

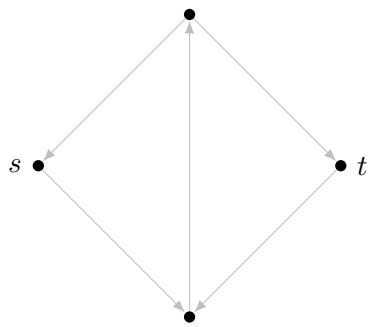
So max-flow cannot be solved by a simple greedy algorithm where you keep using up flow paths until none are left. So we need to be slightly smarter. But Ford and Fulkerson realized you only need to be *slightly* smaller.

Definition 8.12. Given a graph G with capacities u and flow f, we define the residual flow as

$$u_f(v,w) = u(v,w) - f(v,w)$$
$$u_f(w,v) = u(w,v) + f(v,w).$$

The point is that we want to represent one of our options is to *remove* flow from the edge (v,w); and that is represented by the capacity you have in the opposite direction. Subtracting a certain amount of flow from (v,w) can be seen as adding some now available flow from w to v.

In the above example, the three edges we've traversed would disappear; the original edges that were not touched are still present. However, in the residual graph, the *reverse* edges get the capacity that was lost. So we actually draw reverse edges.



And what's nice is that this graph *does* have a flow path; this flow path is what's known as an *augmenting path*, which just means that it has positive capacity in the residual graph.

It turns out augmenting paths are the key to cracking max-flow. If we have an augmenting path, then we can increase the flow value — we take that augmenting path, and notionally send flow along it. What does that actually mean (note that we've created edges that didn't exist)? Sending flow on a reverse-edge is equivalent to *removing* flow from the forwards edge. So the augmenting path we'd perform here would add flow to the two edges which haven't changed, but *remove* flow from the edge in the middle. And that's exactly what we wanted to do.

The point is that you're doing this in a way that preserves the balance constraints — adding outgoing flow has the same effect on net flow as subtracting incoming flow. So adding everywhere on an augmenting path (which means subtracting from some reverse arcs) maintains the conservation constraints.

What if there is *no* augmenting path? Well, if there's no augmenting path in the residual graph, then it has a cut of value 0. What does that mean? It means every edge leaving the cut has residual capacity $u_f(e) = 0$ — there are no leaving edges.

Now, why would the residual capacity of an edge be 0? This means every leaving edge has $f(e) = u(e)$. Well, that's interesting.

Let's also think about the *entering* edges; what can we say about those? If there is an entering edge that carries some flow, then the reverse edge is going to have residual capacity (if there's flow on an edge, then you give its reverse edge capacity, to represent taking flow off the edge). But if the reverse edge has residual capacity, then, well, it's exiting the cut, so that's impossible. (We just said that we found a cut with 0 capacity leaving.)

So the conclusion is that every entering edge has 0 flow.

Now when we put these together, we find that the *net flow* across the cut is equal to the exiting capacity of the cut. (We're using all the outgoing capacity, and none of the incoming capacity.)

But we wrote down somewhere that the net flow crossing an s-t cut is just equal to the value of the flow.

So if there is no augmenting path, then the value of the flow is equal to the capacity of this cut that we identified. We've already shown flow can't be bigger than the exiting capacity of the cut; and if there is no augmenting path, then the flow value *equals* the capacity.

So this means the flow is maximum!

So we now have a way of certifying that a flow is a maximum flow. And every cut is an upper bound on the flow value, so the minimum cut is an upper bound on the flow value; a maximum flow is tight at some cut, so that must be the minimum cut. What we've proven is:

> **Fact 8.13 —** For any graph, we have max-flow = min-cut.

What's interesting is that these two quantities kind of certify each other. If you want to send as much grain as possible on your rail networks, that's a max-flow problem. If you want to do the smallest work necessary to destroy the network so that nothing can be sent, that's a min-cut problem. If you have one then you have the other; and if you have both, they prove each other to be optimal. This is known as *max-flow min-cut duality* — these problems are duals of each other (one provides a bound on the other).

To summarize this, we've proven the following.

Proposition 8.14

The following three statements are equivalent.

- f is a max-flow.
- There is no augmenting path in the residual graph G_f.
- The value of the flow is $|f| = u(S,T)$, where for some s-t cut (S,T).

§8.6 Algorithms

Now we have a good sense of how to understand optimality; the next step is to understand algorithms.

Thinking about verification of optimality (or detection of non-optimality) often gives you insight for how to work towards optimality. So given what we've just done, what's a natural approach to finding a maximum flow? Well, we've just described a mechanism for detecting when a flow is not maximum — that means there is an augmenting path in the residual graph. So what should we do to improve our flow towards being maximum? Well, we can just add such a path.

Algorithm 8.15 (Augmenting path algorithm)

While the flow is not maximum:

- Find an augmenting path in the residual graph.
- Augment the flow by this path (by the minimum residual capacity among edges on that path).

This is the right amount to augment by because you can do it. It has the nice effect of destroying some residual edge. (It also creates residual edges, so it's not clear that you're making progress, but we'll show you are.)

First, how fast can we find an augmenting path in the residual graph? You can do this in linear time, i.e., $O(m)$, using DFS or BFS or whatever FS you want. So we're well on our way to having an algorithm for max-flow; it remains to consider how many times we need to do this.

§8.6.1 Number of augmentations

Question 8.16. How *many* of these augmenting paths will we have to find if we want a max flow?

Let's creep up on that. First, suppose all capacities are 0 or 1. Then the fact that there are m edges means that there are at most m paths in the decomposition, so the value of the flow is at most m. Alternatively, there are at most m edges in the minimum s-t cut, which also proves the same thing (since max-flow equals min-cut).

On the other hand, every augmentation increases $|f|$ (the value of the flow) by 1. So we can conclude that m augmenting paths suffice to find a max-flow.

We can actually refine this. In a so-called *simple* graph (one with no parallel edges), the same arguments show that the flow value is at most n (you can only have $n-1$ paths leaving s, because each uses an edge out of s; alternatively, the cut with just s on one side has value at most $n-1$). So $n-1$ augmenting paths suffice.

Let's generalize this to integers.

Question 8.17. Suppose your capacities are integers upper-bounded by some number U. What can we say about the number of augmenting paths that we need?

In this case, we can say the value of the max-flow is at most mU (everything in the above arguments just multiplies by U), or nU for simple graphs. And again, each augmenting path finds some integral value (which is at least 1). So mU (or nU) augmenting paths suffice.

Interestingly and importantly, our first augmenting path is going to involve integral capacities, so we subtract an integer amount from each edge, which means the next round capacities will still be integral. And this remains true. So we get for free the following interesting lemma:

Lemma 8.18

If we have integer capacities, then we have an integral max-flow (meaning that the flow on every edge is an integer).

And each augmenting path takes time $O(m)$ to find, so we get an $O(m^2U)$ algorithm (or $O(mnU)$ for simple graphs).

Is this polynomial? It doesn't satisfy our typical notion of polynomiality, because numbers of size U can be written down in only $\log U$ bits. So this runtime is actually exponential in terms of the number of bits you need to write down the numbers. This is called a *pseudopolynomial* algorithm (you can define that in a number of ways, but one is to say it's polynomial in the size of the graph and the *values* of the numbers). It's polynomial if all the values are small (e.g., 1), but for larger numbers polynomiality breaks down. That'll be our next challenge for Wednesday — developing algorithms that are truly polynomial-time, regardless of the size of the numbers.

§9 September 25, 2024

§9.1 Recap

Today we'll talk about max-flow algorithms. Last time we got a good understanding of the structure of maximum flows, and we talked about the starter problem of determining whether a given flow is maximum. This took us to the max-flow min-cut theorem, as well as the notion of *augmenting paths*. Putting these together, we said that a flow is a maximum flow if and only if there's no augmenting path in the graph, if and only if there is a saturated s-t cut. The value of the cut is the value of the flow; the value of any cut is an upper bound on the value of any flow and vice versa, so when we have equality, we know we've got the max-flow and min-cut. This is our first example of *duality*.

§9.2 Augmenting paths algorithms

From the fact that an augmenting path is guaranteed if and only if our flow is not maximum, we got to our first algorithms for max-flow, namely augmenting path algorithms. Here the general idea is that you repeatedly find an augmenting path and saturate it, until there aren't any left. The most obvious way to find an augmenting path is through BFS or DFS or something similar; this takes $O(m)$ time. All that's left to understand the algorithm is to know how many times you repeat this loop before you find your maximum flow.

Last time we talked about the case of integral capacities — we said that with integral capacities, your first augmentation will be by an integer, which keeps all capacities in the residual graph integral. (All the augmenting path-finding is done in the residual graph; that's our third big idea from last time.) So by

induction, you'll always be finding integral augmenting paths; this means your optimum flow is going to be integral. It also means you find at least a one unit increase per iteration, which means your overall runtime is $O(mf)$ (where f is the value of the flow). And we talked about various ways to bound the value of the flow. If you have a unit-capacity graph with no parallel edges, then there's a cut of value $n-1$ (if you just consider the source as one side), so there's an upper bound of $n-1$ on the value of the flow, giving an $O(mn)$ time bound for simple graphs. Even with parallel edges, every unit uses up an edge, so you get a bound of $O(m^2)$.

Then we considered, what if there are capacities, up to some integer U? In the standard max-flow problem, when you have capacities, there's no reason to think about parallel edges (because you can combine them). If you have capacity U per edge and no parallel edges, then you have at most nU flow coming out of the source, so you get an upper bound of $O(mnU)$.

We talked about the fact that this is polynomial in the quantities m, n, and U; however, it doesn't fit the classic computer-science notion of polynomiality because it's not polynomial in the size of its input. It's polynomial in the size of the *graph*, but when we look at the numbers on the edges, you can have an n-bit number without changing the input size, but that gives you a capacity of 2^n, and now this bound becomes 2^n (which is not polynomial).

§9.2.1 Rational capacities

Now we'll talk about some other limitations of this algorithm. So far we've only been considering integers, but what about *rational* numbers? Well, algorithmically it's no big deal to think about how to deal with this (we can do rational arithmetic instead of integer arithmetic). All this does is we'll be generating new fractions at various times; we don't know exactly what fractions we'll end up with, but they *will* share a common denominator with the ones we started with.

So we can first multiply through by all the denominators, to make everything an integer, and then scale down by this product in the end.

With rationals, we assume all numerators and denominators are at most U. Then we imagine multiplying by all the denominators, applying the integer algorithm, and then dividing by all the denominators. That will work, right?

In fact, you don't actually have to do this — you can run the algorithm with rationals, and it'll implicitly do everything in terms of the common denominator.

What can we say about the runtime if we take this approach? Multiplying through by the denominators shouldn't change the runtime, so we can just think about this case. But if we do this, then we'll have numbers of size U^m. So you take a graph where the largest number involved is U, but when you multiply by the denominators, you end up with numbers of size U^m; and now you're looking at a runtime of $O(mnU^m)$. This is clearly not polynomial. (Even if you don't multiply through by the denominators, the addition of fractions could produce such tiny augmentation values that you're augmenting by U^{-m} in every iteration, and it'll still take you U^m time.)

So already with rationals, we're starting to run into trouble with this naive algorithm.

§9.2.2 Real capacities

What if the capacities are *real*? If we talk about computing with reals (with full precision — talking about reals means we want the exact answer, or else you could turn them into rationals), this could take up to infinite time, just by generalizing the argument about rationals — here there *is* no common denominator, and you can worry that you'll end up with smaller and smaller augmentations, with no real lower bound (so that you never actually get enough).

So for reals, the situation is bad.

But it's actually worse! Maybe we never actually get to the answer, but what about the limit? You might hope that the augmentations get smaller and smaller but only close to the optimum, so that in the limit at least you end up with the max flow. But this is false — you can construct max-flow problems where the generic augmenting paths algorithm comes up with a limit that is not the actual max flow. (This is quite cute, and it's weird — what seems like a linear process doesn't have the right limit.)

§9.3 A greedy improvement

So we have this problem that augmenting paths works for integers but could be slow, and falls apart for rationals and reals. What do we do?

The flexibility in our augmenting paths algorithm is that it just says 'find an augmenting path.' But maybe we can be more careful about *which* augmenting path we find, so that we do better than the generic algorithm.

We're trying to find the maximum flow; what's the greedy approach to an augmenting paths algorithm (that tries to get there as quickly as possible)? We can try to choose an augmenting path with the maximum capacity (where by the capacity of a path, we mean the capacity of the bottleneck edge — i.e., the minimum capacity of an edge on that path). So we would like to find the path whose minimum capacity is maximum, and augment along that — that's the natural greedy thing to do.

§9.3.1 Finding a max-capacity augmenting path

Now we have two questions. The first is how — how do we *find* a maximum capacity augmenting path?

There are two nice answers here. One is that this is a very Djikstra sort of problem — it's kind of like shortest paths, except that the length of your path is defined by the minimum edge on the path, not the sum of edge lengths. In fact, that makes it kind of like a minimum spanning tree. This is in a directed graph, though, so a minimum spanning tree is not defined; there is a notion of a minimum *directed arborescence*, and that's kind of what you want here.

And you can compute this like Djikstra, or more like Prim — it's basically Prim's algorithm but for a directed graph. You have a certain set of vertices that you've reached, which defines one side of a cut; and then you have the edges that cross that cut, and you want to take the largest edge crossing the cut, which brings another vertex into what you've reached. And if you do that n times, you reach the entire graph (including the sink). And you can use a heap to keep track of the largest edge crossing the cut, just as we do with Prim (or Djikstra).

So we know how we can solve this, in $O(m + n \log n)$ time (or $O(m \log^* n)$ time, or $O(m \log \log^* n)$ time, or all those wacky bounds we had for Fibonacci heaps).

Suppose that you did not know Fibonacci heaps; what's a more simplistic approach that still gets good time bounds? A simpler approach is to do a binary search on the value of the bottleneck edge. You want a path that is composed only of edges whose capacity is at least the bottleneck, and you're asking, what's the largest bottleneck I can find? So you do a binary search — you set your threshold, throw away all edges smaller than the threshold, and see if you can get from s to t. If you can, you've found a path whose capacity is at least your threshold, and you take a larger threshold. If you can't, then you take a smaller threshold. So you do binary search on the threshold (there are m possibilities), and do a BFS to find a path; this takes $O(m \log m)$ time. (This would be easier to implement in practice.)

§9.3.2 How many iterations?

The next question is how many iterations we'll need. So we need some way to relate the amount of flow we're able to send in one step to the value of the maximum flow. And a good way to do this is to go back to thinking about flow decomposition.

Flow decomposition tells us that you can decompose any flow into at most m paths (and cycles). Let's suppose that the flow in the residual graph has value f. (We haven't emphasized this before, but if you find the max-flow in the residual graph and add that to whatever flow you already have, then you have the max-flow of the original graph — the max-flow of the residual graph saturates a cut, and when you add in the old flow, that also shows it's the max-flow of the original.)

And if the flow in the residual graph has value f, but it's made of only m paths, then there has to exist a path with value at least f/m — there's f units of flow travelling in total, and it's spread out along at most m paths, so some path has to have f/m flow on it. (The cycles don't contribute to the value of the flow.)

And if some path in the residual graph has at least f/m flow on it, that path has to have a bottleneck edge that's at least f/m. And from that, we can conclude that the max-capacity augmenting path (generally abbreviated as the maximum augmenting path, or MAP) has capacity at least f/m.

So if there is f flow yet to find, then one iteration of MAP (finding a maximum augmenting path) finds at least f/m of it.

Now we're in great shape, because we can iterate this argument — we start with f residual flow and remove f/m of it, so we've multiplied it by $1 - 1/m$. This gives the following result.

> **Claim 9.1 —** If the max flow is f, then t rounds of MAP reduce the residual flow to $(1 - 1/m)^t f$.

§9.3.3 Integer capacities

Now to apply this, let's suppose we have integer capacities at most U. The above bound is not immediately applicable because we never reached *zero* residual flow. But we don't have to get to 0 to know we're done — we can rely again on the integrality property, that as long as we keep all our augmentations as integers, all residual capacities are integers. So if we can get the residual capacity to be *less than* 1, then it's an integer less than 1, so it must be 0.

So we just drive the residual capacity below 1, using integral augmentations. And that implies it's 0, and we have a max flow.

How long does this take? We need to arrange for

$$\left(1 - \frac{1}{m}\right)^t f < 1,$$

which means $(1 - 1/m)^t < 1/f$. Throwing in our favorite approximation $1 - x \approx e^{-x}$, this becomes $e^{-t/m} < 1/f$, which means we want $t = O(m \log f)$.

> **Student Question.** *What are the requirements for the approximation $1 - x \approx e^{-x}$ to work well?*
>
> **Answer.** It works well as long as x is small; if we care just about asymptotics, it's enough for x to be bounded away from 1.

Now we recall from before that $f \leq nU$, so we get $t = O(m \log nU)$.

That's our bound on the number of iterations we need. And now we plug in the time for doing a single iteration, which is about m. Now we're going to get a bit sloppier and use $\widetilde{O}$ to say that we're not worried

about log factors — $\log m$ and $\log n$ aren't interesting when we're talking about algorithms with runtimes of n^3, for example. (We do want to keep track of U, though.) So we get a runtime of

$$\widetilde{O}(m^2 \log nU).$$

Unlike the first algorithm we described (which was pseudopolynomial — not polynomial in the size of the input), this *is* polynomial in the size of the input — if you want to deal with numbers of size U, you're going to need $\log U$ bits to represent those numbers, so you're allowed to have runtimes polynomial in U. From a theoretical perspective, this is a huge improvement; it's also an improvement in practice, since we often work with large problems.

§9.3.4 Rational capacities

What if we go to rationals? We saw we deal with them by multiplying through by the denominators; this has the effect of turning our maximum capacity from U to U^m. Does that pose a problem here? It actually doesn't, because it's inside the log, so that's okay — we get

$$\widetilde{O}(m^2 \log nU^m) = O(m^3 \log nU).$$

So we're polynomial even with rational numbers (which includes the floating point numbers we work with in practice).

But this still doesn't resolve the issue for real numbers — the fact that we're creeping ever closer, with real numbers, doesn't mean we actually get there. And so this draws up a distinction that's made in the theoretical computing literature, between *weakly* and *strongly* polynomial algorithms. (This is *not* a strongly polynomial algorithm.) The definition of a strongly polynomial algorithm is that the runtime is not dependent on the numeric values.

Lots of optimization problems we work with have a structural part (here, the graph) and a numeric part (here, the numbers we sprinkle onto it). A strongly polynomial algorithm has a runtime only depending on the structural part (if you are allowed to do addition and subtraction of arbitrary real numbers in constant time, without worrying about precision).

Instead, this algorithm is what's known as *weakly* polynomial. But this *is* polynomial according to the input-size notions from complexity theory, so this is certainly an accomplishment.

There's an obvious next question:

> **Question 9.2.** Is there a strongly polynomial algorithm for max-flow?

We're not going to see this yet. Instead we'll see another weakly polynomial algorithm, which is nice because it introduces a very cool and useful trick called *scaling*.

§9.4 Scaling algorithm for max flow

This algorithm was discovered in the US by Gabow in 1985, but it had already been developed by Dinitz in 1973 in Russia (the problem was that people in the US didn't know how to read Russian).

The idea of scaling is that if we're dealing with numbers represented as a sequence of bits, well, each bit is a 0 or 1. So if we could just concentrate on one bit at a time, we'd be back in the case of having no numbers at all, just 0-1 capacities. And that would make life simple — we saw that there, the naive augmenting paths algorithm (where you just find arbitrary augmenting paths) works fine.

(Here we're working in the context of integers; we can think about rationals later.)

So that's the idea of scaling — to apply the unit case to general numbers. How do you do that? We'll first describe it in a sort of recursive fashion, though we'll give our specific algorithm in a non-recursive way.

What we'll do is that we first round down the least significant bit of our input — we take every number, divide it by 2, and ignore the fractions. We've just simplified our problem; let's solve it.

Now we've got a solution; let's un-round both the inputs and the solution.

With max-flow, you can imagine we round down the capacities, solve the problem, and then un-round by bringing back that lost flow. We can also double the flow (we divided the capacities in half and found a flow; when we multiply them back by 2, we can multiply the flow by 2).

The claim is that this is pretty close to optimal, and we just need to 'fix up' the solution. What does that mean?

With max-flow, we've rounded down the last bit of capacity and found a max flow. So suppose we've got a rounded solution, and now we un-round — we double the capacities and the flow (which changes nothing). But what's the other part of the un-rounding? We have to bring back that least significant bit. What that means is we need to add 1 to some capacities.

> **Question 9.3.** When you un-round, what happens to your solution — is it still optimum?

The answer is 'not necessarily,' because you've added some flow to the graph. So we have a non-maximum flow, and we want to make it maximum. And what algorithm do we use for that? Well, we use augmenting paths.

> **Question 9.4.** How many augmenting paths are we going to need to use to fix up this flow?

Well, all the 0 arcs in the residual graph for the rounded solution have capacity at most 1 in the residual graph for the unrounded solution (because we double them, which does nothing, and then maybe add 1). (By 0 arcs, we mean saturated arcs.) That's only talking about single edges; but what does that tell us about the total amount of flow we might have to send through the network?

We had a max-flow in the rounded graph, which means there was some cut of value 0 in the residual graph. Note that this may not have been the cut separating s from everything else — it could be a cut with m edges. But it can't be a cut with more than m edges, because there's only m edges. So the min cut of the rounded graph has residual capacity 0 in the rounded solution (by max-flow min-cut), which means it has capacity at most m in the unrounded solution (because each of those arcs must have had capacity 0, and now they have capacity at most 1).

And that tells you m augmenting paths suffice to fix your solution.

What does that mean for runtime? Well, what sort of augmenting path do we need to implement? What's nice about scaling is we're just thinking about the unit capacity case, so we don't need to do anything fancy (e.g., finding a maximum augmenting path); we just find any maximum path (each has capacity 1, because everything is an integer), and we can do this by BFS.

So we need to do just m BFS's; this means it takes $O(m^2)$ time to fix up the almost-optimal solution.

How many times are we going to have to do this? We do one iteration per bit, so the runtime is $O(m^2 \log U)$. You could implement this recursively, but another way to say it is that you start with all-0's and roll in one bit at a time, starting with the most significant.

This is deeper conceptually but probably much simpler to implement (you're not using any fancy data structures, just BFS). The runtime is also a bit better — here the runtime only depends on $\log U$, while for max-capacity augmenting paths, the runtime involved $\log f$ (and f can even be nU).

There are scaling algorithms for lots of things (e.g., shortest paths, min-cost flow which we'll see next week), but this is a very nice application.

Student Question. *What if we applied scaling for rationals or reals?*

Answer. For rationals we know what happens — the numbers increase by a power of m, so we'd have $\log U^m$; but that just puts another m factor on the outside, so it's still polynomial.

But suppose you have real numbers and you run for a certain number of iterations; can you say anything about the error? You could back up to saying you're using a certain number of bits, so you're approximating reals by rationals and you can apply the reasoning for rational numbers.

But there's a more direct thing you can say, which is a generalization of what we said here. If we stop at a certain point, what's the amount of error? Well, we have a cut of capacity 0 in the residual graph so far. To understand the error, we can take that graph and roll in all the infinitely many remaining bits at once. The edge was 0, and what it gains is just the tiny bit of numeric value in the bits I'm rolling in. If I brought in 50 bits, then what's left is just 2^{-50}; this means I can add at most 2^{-50} to each edge crossing the min-cut, so the total error is at most $2^{-50}m$.

Remark 9.5. David will be away on Friday; he'll see how far we get to decide whether we do a video thing or just cancel the lecture.

§9.5 Strongly polynomial max-flow

We've now seen two weakly polynomial algorithms for max flow; what about strongly polynomial algorithms?

Strongly polynomial algorithms will require a new idea. Everything we've done so far relates the progress of the algorithm to the amount of flow we still have to find — we said the maximum augmenting path gets a certain fraction of the max-flow. But with real numbers, you can get a fraction but never get all the way there. (The maximum augmenting paths algorithm is known as a *primal* algorithm — it only looks at the value of the optimum as its guide, and works towards improving its portion.)

So to get a strongly polynomial algorithm, we need a different notion of progress towards the optimum. We'll develop a *primal-dual* algorithm, which looks at the dual as a source of information about the primal.

What allows us to certify that we've reached a max flow? The max flow was certified by an empty cut in the residual graph. (We've been trying to say s-t cut, but we're only talking about s-t cuts when we say cut.)

Having an empty cut in the residual graph tells us 'yes, this is a max flow.'

Question 9.6. How can we creep up on the min cut?

We can talk about the capacity of the min cut going down, but this doesn't actually change the approach (it's the same notion as before). So we need a different notion of something more and more like an empty cut.

What we'll focus on is that an empty cut is a demonstration that it's not possible to get from s to t. What might be a partial result in that direction, saying that it's *hard* to get from s to t? The answer is distance. A cut of value 0 is essentially creating an infinite distance between s and t. Maybe instead of reducing the residual flow, we can aim at *increasing the distance from s to t* until it becomes infinite.

How many times do you have to do this? Only n times — distance is measured in edge counts, and the longest shortest path in a graph has only $n-1$ edges on it. So if we can get the shortest path up to n, then it's actually ∞ and we're done (this means we have an empty cut between s and t).

Here's another related insight. We talked about flow decomposition, and said every flow decomposes into at most m paths. But not all paths are the same. If we assume we have a unit-capacity graph, and the distance from s to t is k, there can be a cut of value m; so if we don't know anything else about the graph,

we can say the flow might have value m (if there are many parallel edges). But if the distance is k, we can say more — every flow path uses k capacity, which tells us that the residual flow is at most m/k. So this also reflects the notion that if you can get the distance up, then you're getting closer to optimum.

§9.6 Shortest augmenting path algorithm

We'll use this insight to get the *shortest augmenting path algorithm* (developed by Edmonds and Karp in 1972). We're going to look for the *shortest* augmenting path. This is still an augmenting path algorithm, so it has the same bounds as before, but we'd like to say more.

First, how do we *find* a shortest augmenting path? We can just use BFS; this is actually easier than max-capacity augmenting paths (we don't need a data structure), and we don't need scaling. (This is one of the nice things about this perspective — the primal problem involves numbers, but the dual problem is just about edge counts, so it's discrete; and discrete is always better than continuous.)

So we can *find* a shortest path easily; the question is, how many do we need? It's not obvious that we are making progress — maybe when I find a short augmenting path, it actually brings s and t closer together. But it turns out that's not true, and we can prove it.

First, the intuition behind *why* you should look for a shortest path is that if you augment along a shortest path, you destroy that path. This is a good thing — you need to destroy the shortest path in order to make s and t further apart. The worry is that by destroying that path you create others, or shorter ones; and we're going to prove that doesn't happen.

> **Claim 9.7 —** Under shortest augmenting paths, no vertex ever gets closer to s (or t).

We'll also show that s and t get *farther* apart, but for starters let's just show that nothing gets closer.

Proof. Assume we have distances $d(v)$ from s before we run shortest augmenting paths, and $d'(v)$ after. And let's suppose that some vertices get closer, meaning $d'(v) < d(v)$ for some vertices.

Let v be the *closest* to s among these vertices afterwards — i.e., v has the minimum $d'(v)$ (among the vertices for which $d'(v) < d(v)$).

So we've got s, and some shortest path that gets to v (under d', i.e., after augmentation). Let w be the preceding vertex on this shortest path — i.e., the one immediately before v.

Then we know $d'(v) = d'(w) + 1$ (because we took a shortest path). Meanwhile, we know $d'(w) \geq d(w)$ — this is because we chose v to be the closest vertex among those that got closer to s, which means w did not get closer.

> **Claim 9.8 —** The edge (w, v) had zero capacity before the last augmentation.

In other words, we're claiming this shortest path did not previously exist, because this edge did not exist. What would it mean if this path did exist? Then v couldn't have gotten closer. We can formalize this by putting some math together.

Proof. If the edge (w, v) did exist before, then we could say that

$$d(v) \leq d(w) + 1,$$

because we could have gotten to v by going to w and then taking this edge. But we also said that $d(w) \leq d'(w)$, which means

$$d(v) \leq d(w) + 1 \leq d'(w) + 1 = d'(v).$$

So v did *not* get closer, which is a contradiction to our definition of v. □

So (v, w) did have zero capacity before the last augmentation. Now, if it had 0 capacity before the augmentation, why is it here now — where did it come from? Well, the only way to create capacity (which is what we just did) is if the augmenting path went through the *reverse* edge (v, w).

So our augmenting path went through (v, w). And it's a *shortest* augmenting path, because that's the algorithm we used. And that means the shortest path to w (before the augmentation) went through v. And that says $d(w) = d(v) + 1$ (before augmentation).

But if we now put that together, we find that

$$d'(v) = d'(w) + 1 \geq d(w) + 1 = d(v) + 2.$$

And this tells us v actually got *farther* away from s, so it certainly didn't get closer (which is what we assumed); so we have our contradiction. □

So shortest augmenting paths never decreases the distance from s to any vertex or t to any vertex; in particular, it doesn't decrease the distance from s to t. That's good but not great — we want to *increase* the distance from s to t.

The next lemma takes care of this.

Lemma 9.9

Within $mn/2$ shortest augmenting paths, we have $d(s, t) \geq n$.

That means this many shortest augmenting paths is sufficient to find a max flow.

Proof. Each flow augmentation saturates an edge. The whole worry about max flow algorithms is that they're not greedy — you may saturate an edge and then unsaturate it, and then saturate it again, and unsaturate it, and so on.

But with shortest augmenting paths, what does it mean for an edge to get saturated again? Well, the edge was gone, and it has to come back; this means we sent flow along the reverse edge.

So if our edge is (v, w), then once we've saturated the edge (v, w), we can't use it again until we send flow along (w, v).

And if we send flow along (v, w), that tells us that $d(w) = d(v) + 1$. Meanwhile, if we send flow along (w, v), this means $d(v) = d(w) + 1$. But these numbers are only ever increasing — we just proved that. So you're in a situation where w is farther than v; and w cannot get closer, so the next step requires v to be farther than w, so that requires v got farther.

So you cannot saturate w again until after v has gotten farther than it was before (in fact, it has to get 2 steps farther).

And this is the end — to saturate an edge again, you need $d'(v) \geq d(v) + 2$. This means you can only saturate an edge $n/2$ times before the distance to that edge is bigger than n (and if the distance to the edge is bigger than n, you can't reach it, so you can't saturate it).

So this saturates every edge at most $n/2$ times; there's only m edges and one has to be saturated on each augmenting paths, so we're done. □

And that completes the algorithm — a single step takes $O(m)$ time, and we only need to do $O(mn)$ augmenting paths, so we get an $O(m^2n)$ time algorithm. This has no dependence on the numbers, so it's strongly polynomial.

As you can see here, the runtime for this algorithm is worse in terms of m and n than the weakly polynomial algorithms. This is typical — you're asking for more, so you end up with a worse time bound (unless the numbers are really gigantic). In practice, you'll usually want to use the weakly polynomial ones.

There's a smarter way to do this kind of shortest paths destruction called *blocking flows*, that leads to some better runtimes; David will decide whether to do it in video on Friday or in-person on Monday.

§10 September 30, 2024

David thinks the room is too cold, and will talk to facilities about increasing the heat.

Today we're going to finish up max-flow; once we've mastered that, we'll make the problem harder.

§10.1 Review

Last time, we finished looking at the shortest augmenting paths algorithm. We showed that the shortest augmenting path was a good dual algorithm for optimizing a flow, not by directly pursuing the value of the flow objective, but instead by pursuing a different objective kind of tied to the max flow — we instead try to maximize the distance between the source and the sink. A shortest augmenting path was the natural way to do this. And we showed that you can find one in $O(m)$ time using BFS; this gives you a shortest augmenting path (there's no lengths to worry about, we're just counting the number of edges). And we showed that a shortest augmenting path doesn't decrease the s-t distance; and in fact, after we do enough of them, we actually *increase* the distance. So we showed you can only have about mn shortest augmenting paths before the distance reaches n, which is effectively infinite (it means there's no actual path from the source to sink). So if you can find one in $O(m)$ time and $O(mn)$ of them suffice, this gives an $O(m^2n)$ time — strongly polynomial — max-flow algorithm. This was a nice achievement — to get a strongly polynomial time bound.

But it's still a bit unsatisfactory because in a simple unit capacity graph, it only took us $O(mn)$ (or $O(mf)$) time. So it feels like maybe we're a bit far off. We also had a scaling algorithm with better runtime. So we're sacrificing something for our strongly polynomial time bound.

Today we're going to try to improve on this by being smarter about our approach to increasing the distance from s to t.

§10.2 Motivation

We're doing a BFS, which tells us where the shortest augmenting path is; and then we augment along that shortest augmenting path. And then we do it all again — we do another BFS, and that'll tell us the new shortest augmenting path.

This feels like a waste of effort — if I just did a BFS to find the shortest augmenting path, then where's my *next* shortest augmenting path going to need to be? We did a BFS to just find one shortest augmenting path, but the whole rest of the BFS is still valid. So it feels like a lot of the time, we ought to be able to find our next shortest augmenting path without redoing all that BFS work. We may be unfortunate, in that the one path we found destroyed the key edge that reaches everything else; but a lot of the time we might expect to be able to reuse that BFS work. And a common theme is that if you're going to do work, you should make it count — you should get as much flow as you can out of it.

§10.3 The blocking flow technique

The idea is to use one augmentation to destroy *all* the shortest paths in a graph. If we destroy *all* the shortest paths in a graph, that's going to mean that the s-t distance increases *immediately*; and that will mean that n of these blocking flows suffice to reach the max-flow.

What are these blocking flows?

Definition 10.1. The admissible graph is the graph constructed by a BFS of the residual graph.

It consists of layers of vertices by their distance from s. So we start with s, then start the BFS and find a whole bunch of vertices at distance 1 from s. We continue the BFS, and get a new layer of vertices at distance 2 from s. And we continue this process for as many layers as we need in order to find our way to t at the other end of the admissible graph. We might not actually visit all the vertices, since once we reach t we've got the shortest path (there may be other vertices further away, which we won't pay attention to).

Definition 10.2. The admissible edges are the edges which go forwards (i.e., from layer i to $i+1$).

Where are the other edges of the graph? Certainly there could be some backwards edges, going just about anywhere (as we do BFS, we'll find some edges pointing to vertices we've already seen; of course these are not shortest path edges, and they don't matter). These are called *back* edges. We might also have *vertical* edges, that go to the same layer. These also are not shortest path edges, and don't matter.

Note that we cannot have edges that traverse more than one layer — if we had an edge that traversed more than one layer, then we put the second vertex in the wrong layer. So all we have is admissible edges that move exactly one layer forwards, and other edges that stay in place or move backwards. And this is the shape of the admissible graph.

Given this admissible graph, what can we say about the shortest paths in our residual graph? They're any and all paths from s to t that only use admissible edges. If you have a path from s to t where every single edge moves one step forwards, then every such path has the same length; and they're all shortest paths.

Definition 10.3. The admissible paths are the paths of admissible edges from s to t.

Fact 10.4 — A path from s to t is a shortest s-t path if and only if it is an admissible sS-t path.

So the BFS doesn't just find you *one* shortest path; it kind of encodes *all* the shortest paths.

Once we have this notion, we can define a *blocking flow*.

Definition 10.5. A blocking flow is a flow that saturates at least one edge on *every* admissible path.

Notice that this blocking flow might not be a max-flow — there might still be residual paths from s to t after you've put this flow on. But they won't be admissible paths. So this is a kind of 'maximal' flow, but not necessarily a 'maximum' flow. And this difference is important to let us compute it much more quickly.

Example 10.6

In a unit capacity graph, a blocking flow is a maximal collection of s-t paths (you can greedily pile in s-t paths, and once you can't pile in any more, you have a maximal flow, and therefore a blocking flow).

Student Question. *If we just took the admissible graph, is a blocking flow the max-flow of the admissible graph?*

Answer. David is inclined to say yes but would need to think about it longer.

The key claim is the following.

Claim 10.7 — Adding a blocking flow increases $d(s,t)$ (the distance from s to t).

Proof. The blocking flow kills at least one edge on every admissible path. This is nice — we've destroyed all the shortest paths that we can see in the admissible graph. But does it create any *new* admisslbe paths? That's something we have to worry about — any path we can see right now can no longer carry flow, so those paths from s to t don't exist, but maybe we've created a new shortest path.

But this can't happen. The key insight is that augmentation does create new residual arcs (which is what we have to worry about), but these arcs all go *backwards*. This means any new s-t path must use either vertical or backwards arcs. (Even without the new arcs, it's conceivable that there is still a path that uses vertical arcs.) This means there's no path that uses *only* forwards arcs — every path must use a vertical or backwards arc. And that counts as either a wasted step or a step backwards. If there's k layers, then the old shortest path had length k, and the only way to get from s to t in k steps is to go forwards on each step (because there are no multi-layer forwards edges — once you lose a step, you're never going to make that up, so you'll never be able to get to the sink in time to match the old shortest path). □

This can also be used as a different way to analyze shortest augmenting paths — we made an argument about distances increasing, but you can use this to make a more pictorial version of that argument if you think about a single path and what happens to the vertices on it. So in a way this is more intuitive than that algorithm.

§10.4 Finding a blocking flow

Now we're in good shape. Each blocking flow increases the distance, and we know when the distance is n we're done. So we just need to find n blocking flows to find the maximum flow. So the only remaining question is:

Question 10.8. How do we find a blocking flow?

§10.4.1 The unit capacity case

Let's start with the unit capacity case (this is always a good idea). What's a natural way to try constructing a blocking flow? Well, let's think greedy. The greedy appraoch is to just start at s and start marching forwards until you get to t. Now you've found *a* path, and you can do it again.

So we start a DFS in the admissible graph; we advance along this DFS until we reach t. And when we do, we say, 'hooray, I found a path of flow; I will augment on it.' And since this is a unit capacity graph, this is going to immediately destroy all the forwards arcs that were used, so they're not in the admissible graph anymore. We'll call this a *block* step, since you've blocked part of the admissible graph by destroying edges.

If we were talking about *maximum* flows, we'd be responsible for creating reverse arcs. But here they aren't admissible, so we don't have to worry about them. This is why blocking flows are great — they let us only think forwards, and not worry about the side effects of the construction we're doing.

So if we reach t, then we've found a path, and we can augment and block that path. Once we've done this, we can start over — we can go back to s, and find another path.

Obviously this is not going to keep working forever; eventually we're not going to succeed in finding another path. Of course at some point we might have a blocking flow, but the greedy process could actually get

stuck sooner. We're destroying edges in this admissible graph, so we might hit a dead end at some vertex v — we reach v through the greedy process, but now there are no arcs leaving v. What should we do in this case?

The obvious thing to do is to back up — if you hit a dead end, you back up and go somewhere else. But there's something else that's valuable and important. Once we know v is a dead end, we know there's no point in going to v ever again. So we can delete v from the admissible graph, which means we can also delete all the edges that are entering v. We're not actually going to be *that* agressive. But we *will* back up to the previous w, and also delete the edge $w \to v$. Because if we get to w again, there's no point in proceeding from w to v, because we already know v is a dead end (and once there is no path from v to t, there never will be one — we never create admissible arcs).

> **Remark 10.9.** Everything is centered on finding a blocking flow here. Once we find a blocking flow and augment, v may come back to life. But for this blocking flow, once v is dead, it's dead. (There are no admissible paths through it; there may still be longer paths.) The point is there are multiple phases of blocking flows, and each phase is its own thing where we're just focusing on destroying the admissible graph.

We call this a *retreat* step — you basically retreat and burn your bridges behind you. And we also have *advance* steps. (Once we back up to w, of course we continue our DFS, trying another arc; eventually we'll delete all edges out of w, and then w becomes a dead end, and we start deleting edges into w.)

This whole thing, running as a loop, defines the canonical blocking flow algorithm — you advance along arcs until you hit t (in which case you augment and block) or hit a dead end (in which case you back up and delete the edge).

> **Remark 10.10.** When we discover v is a dead end, we *could* delete all the edges into v. But we're not going to bother for the sake of analysis (which is simpler this way); we only delete the edge we retreat along.

Now we have an algorithm, and we can start analyzing runtimes. We have three operations: advances, retreats, and blocks. And all of our work is in these three operations, so we just need to total up the work from all of them.

We're going to start by asking about blocks. How long does it take to do a block? The length of the path, which is at most n. So this takes at most n per block. And if we're still dealing with a simple unit graph, then there's at most n blocks — each destroys an edge coming out of the source. If we don't have a simple graph, then there's at most m blocks.

We can actually be a bit stronger than that — what if the value of the flow is very small? There can't be more blocks than the flow — each block is an augmenting path, so it adds flow from s to t. This means there are at most f blocks, so this takes $O(nf)$.

What about retreats? Every retreat destroys an edge, so there are m retreats.

And what about advances? This could be annoying to analyze, but we don't have to analyze it. Instead, we'll connect it to something else. When you do an advance, that's going to be followed by something else happening to that edge. You'll eventually delete it, either by a retreat or by a block — after we advance, we'll either do a block through that edge, or a retreat. So the advance work is just equal to the block work plus the retreat work.

And then we're done; the overall runtime here is $O(m + nf)$.

But also, a second argument about blocks is that if a block uses k edges, then you delete k edges. So every edge gets blocked just once, which means the blocking work is also $O(m)$ (which is incomparable to nf). For now we'll keep the nf in our back pockets; and overall this shows that a blocking flow takes $O(m)$ time.

And if a blocking flow takes $O(m)$ time, then a max-flow takes $O(mn)$ time. So a blocking flow does give us a good max-flow algorithm for unit capacity graphs.

You might say that we already had a max-flow algorithm for unit-capacity graphs, and it already ran in about this time. But this is actually a slight improvement. This analysis allows parallel edges — we never used the fact that the graph was simple, just that we are deleting edges as we do the blocking flow (and the bound on the number of blocking flow steps doesn't require the graph to be simple). So from that perspective, this is a slight improvement on standard augmenting paths.

But this is the weakest of the benefits; this leads to some very powerful results for capacity graphs, but we'll actually first show how this improves the bounds for unit capacity graphs.

Let's do a slightly better analysis. Suppose we have m edges, and we do k blocking flows. What do we know about the graph after we do k blocking flows? We've increased the distance between s and t — so it's now k. If I give you a graph where $d(s,t) \geq k$, then you can actually say something about the max flow. We have the path decomposition lemma, and every remaining flow path uses up k edges; but that means there's only room for m/k flow.

Now notice that besides increasing the distance, every blocking flow is going to send some flow (specifically, it uses up at least 1 path). So that means m/k more blocking flows find the max-flow.

If we put these two steps together, we can conclude that $k + m/k$ blocking flows suffice in a unit capacity graph. This is true for all k, so what should we set k to be? We want $k = m/k$, so we set $k = \sqrt{m}$ — so $O(\sqrt{m})$ blocking flows find the max-flow. And this means with blocking flows, we improve the runtime for max-flow from $O(mn)$ to $O(m^{3/2})$.

In the worst case of a dense graph, this is still $O(n^3)$, but for a sparse graph this is a dramatic improvement.

There are a number of other bounds that can be proven using the special structure of unit capacity graphs, and we'll do a couple of those on the homework.

§10.4.2 General capacities

Now suppose that we have capacities. What goes wrong with our unit capacity analysis? The general framing of advance/retreat/block still works. But the question is, how much do we pay for them? It's still true for sure that

$$\text{advances} \leq \text{retreats} + \text{blocks},$$

so we don't have to worry about advances. It's also still true that we do at most m retreats, so that's not a lot of work. The problem is blocks.

For the bounds from earlier, we argued that blocking flows took $O(m)$ time, and the reason was that when you did a block, you destroyed *all* the edges on that path; this meant the total block work was bounded by the number of edges. But now a block might only destroy *one* edge on the path.

One way you could deal with this is by saying we'll charge the n block work to that edge — it's at most n work doing the block (updating the capacities), and we charge it all to the edge that got deleted. If we do that, then the total block work is $O(mn)$. This immediately becomes the dominant term in the blocking flow algorithm — it says computing a blocking flow takes $O(mn)$ time, and that leads you to an $O(mn^2)$-time max-flow algorithm.

This is still a noticeable improvement compared to the $O(m^2n)$ time for shortest augmenting paths. It really highlights the benefit of blocking flows. Both of these are essentially pursuing the same goal of increasing the distance. But for blocking flows, we showed that you need mn shortest augmenting paths, and in shortest augmenting paths, finding *each* of those cost you m time. But in blocking flows, the time of finding which paths are shortest is amortized — you compute one layered graph, and then find *many* shortest augmenting paths. And in the layered graph, finding these costs you only $O(n)$, rather than $O(m)$. So you immediately replace one of the m's with an n, just because you made the work done during the BFS count for more.

Student Question. *Where are we constructing the work of constructing the admissible graph?*

Answer. That's also $O(m)$ — each step of blocking flow requires constructing the admissible graph and then doing this. We've shown that doing the blocking flow costs at least $O(m)$, so that swamps the cost of creating the admissible graph in the first place.

§10.4.3 Scaling blocking flows

Now we'll see if we can improve on this a little bit. We also already proved a different bound — in the unit case, we showed a bound of $O(nf)$. We'll break these things apart for reasons we'll see soon: if we only think about the retreats, each blocking flow step spends $O(m)$ on retreats, which means we spend $O(mn)$ time total over *all* the blocking flows. We also argued that each block costs n and adds one unit of flow; this means in total, we're only doing $O(nf)$ block-work. So if we put these together, we get a different bound for blocking flows of $O(mn + nf)$ time (to find a max-flow).

This is a slightly weird analysis, so we should be careful about it. We're analyzing the retreat steps one blocking flow at a time — each blocking flow spends $O(m)$ on retreat steps, which gives $O(mn)$ total. But for block steps, we're analyzing over the entire *sequence* of blocking flows — we do many blocking flows, and each adds some flow to the graph. But in total we're only adding f units of flow; so whichever phases we did that in, we only do $O(nf)$ work.

This highlights that in addition to the $O(mn)$ work *looking* for augmenting paths, blocking paths is smart by only spending $O(n)$ work on each augmenting paths — in contrast to a traditional algorithm, which spends $O(m)$ work on each.

This bound of $O(mn + nf)$ is valid even for capacitated graphs, and if the flow value is small, it's a good time bound.

Are there any techniques that allow us to focus on graphs where the flow value is small? Scaling!

We've already seen the general pattern for scaling — we scale one bit at a time, and for each bit, we update the max-flow based on the new capacity. We found a max-flow in one phase; now we need to roll in the next bit. So we double all the capacities and flow, and now we have to add one unit of capacity to some of the edges based on the new bit that got rolled in.

In our earlier scaling algorithm for max-flow, the key insight was that when we do a new phase and roll in one bit, the resulting graph is special — there used to be an empty residual cut (all the edges had zero capacity), and scaling increases that capacity by 1 per edge, which is at most m. So the new max-flow for this round is at most m.

If the value of the max-flow is at most m, then what's the runtime for the blocking flow algorithm? It's $O(mn + nf)$, but $f \leq m$, so this is just $O(mn)$.

That's for a single scaling step; and that means the *overall* runtime is $O(mn \log U)$. This is compared to the $O(m^2 \log U)$ from augmenting paths with scaling. So once again, we see how blocking flows substitute n (the length of the augmenting path) for m (involved in searching over the entire graph).

Student Question. *Why does the $\sqrt{m}$ bound we found before not work here?*

Answer. That's a great question, and we'll come to it when we wrap up.

§10.5 A strongly polynomial algorithm

Now we have a weakly polynomial algorithm for max-flow with a good runtime. We're supposed to be never satisfied, so the natural question is, can we get a *strongly* polynomial algorithm with this kind of time

bound? Here mn is a natural bound going all the way back to unit capacity graphs — with a simple unit capacity graph, the flow is n and it takes m time to find an augmenting path. And with blocking flows, we've managed to meet that even with capacities, but it's weakly polynomial. So can we be strongly polynomial?

And the answer is yes! In order to do that, let's ask again, what was the bottleneck? The retreats were never a problem; there's always m. The problem is the actual augments — what's making things expensive is that we are spending n work on each of the blocking steps. And unfortunately, only a single edge is being saturated when we do that blocking step.

So what we'd *like* to do is — instead of paying work over the entire path, we'd like to really only work on the saturated edges. Before, work went into augmenting edges that weren't saturated — we had this nice long path and this one edge, and we blocked this path by saturating that one edge, and that took us quite a lot of work because we had to traverse the entire path.

So that took a lot of work, but it didn't actually cause that much damage to our admissible graph. We've done all this work to find an admissible path; what's a nice way to look for the next admissible path? Our two pieces of the admissible path are still admissible, and they still make *most* of a path. So maybe we can start our search from the end vertex of the first piece. And maybe we get lucky and hit most of the remainder. So maybe with just a few short steps, we could re-attach the pieces and get ourselves another admissible path.

What does this call for? It seems like it would be good to have some structure for the data about which paths are available to connect to each other to make a full admissible path from s to t. Let's call it a data structure. So what should that structure be able to do?

We want to keep *pieces* of admissible paths. We'll actually be more precise — these admissible paths may meet. And as we assemble and disassemble many of them, what we'll end up is more a collection of admissible *trees* — trees made of admissible edges. We'll be hunting around; and every once in a while we'll be at some tree and try to advance from its root, and it'll attach us to another tree. And then we have one big tree.

So we want to be able to *link* the root of one tree to a node of another.

Another thing we want to do, when we saturate an edge, is to *cut* a link (to represent the fact that it's not there anymore).

Linking and cutting trees is trivial; it's just adding and removing a pointer. But we *also* want to be able to teleport to the root. We want to be able to do this advance and retreat process; advancing means I try an edge and I've found another tree. But I don't want to waste time exploring the tree; just take me to the root so that I can continue advancing from there rather than wasting all the middle steps. So for any given node, we need to be able to get to its root easily.

But now the hardest thing of all is when we do an augmentation, we need to decrease the capacity on all the edges of the path — meaning we decrease the capacity on all edges from a node to the root. And how do we know how much capacity to decrease by? We need to find the min capacity on the path from a node to the root.

So if we had some amazing magic data structure that could support these operations — a collection of trees where you can ask for the min capacity on a path from the root to the tree, you can ask for the root, and you can attach and cut trees — this would support the blocking flow algorithm.

Back to our advance/retreat/block algorithm, an *advance* is essentially doing a link — we're at some current node and an advance is saying to follow an edge, and then teleport to the head of that tree so that we don't have to waste all the time of the intermediate advances. These advances are again paid for by the retreats and augments.

When we retreat, we're going to have to destroy an edge, which will cause us to cut a tree away from another tree — we can't use that edge anymore, and need to explore a new path.

And in the augmentation step, each augmentation is going to destroy an edge. So there's only m of these. So if we can make each of these augmentations really cheap, then we've made blocking flow really cheap.

This is all we need to do, but it seems quite hard — how do you update the capacities of all the edges on a path in a small amount of time?

When things are hard you need a miracle, so we call the god of data structures. Sleator and Tarjan developed *link-cut trees*, which support all the above operations in $O(\log n)$ time. So no matter how long the path is, you can update the capacities of all the edges on the path in $O(\log n)$ time.

And now that we have this data structure, we can go back and think about our blocking flow algorithm. Your current node will be the root of some tree. When you want to do an advance, you follow an edge from the current node, and you link to the next tree. (When you start the blocking flow algorithm, the trees are isolated vertices; nothing is connected yet.)

When you do a retreat, you cut the edge that you are retreating on, and your current vertex becomes the tail of that edge.

And of course, an augment means you do a find-min to find the minimum capacity — you augment when you have finally reached t, and since you've started at s, when you reach t, s and t are in the same tree. So at that point you find the minimum capacity on the tree path. And you decrease by that, and also cut that min edge.

So each retreat, each advance, and each augment costs you $O(\log n)$ time. And we've shown you only do m retreats, advances, and augments per blocking flow; and so you get an $O(m \log n)$ time blocking flow algorithm, and that gives you an $O(mn \log n)$ time max-flow algorithm. And so you pretty much converge on matching the time bounds for unit capacity graphs, just using a clever data structure with blocking flows.

§10.6 Link-cut trees

How does this data structure work? The details are complicated, but it's easier to think about if you imagine that all your trees are paths. So you want to be able to find the minimum value on a path, attach paths, and cut paths; and you also want to be able to decrease all the values on a path. If it's just a path, it's a sequence of numbers. So you're combining sequences and splitting sequences.

The way Sleator and Tarjan do that is by just putting a big splay tree on top of each path, where each edge is a node in the splay tree. We already know how to link and cut splay trees; we talked about that a couple of weeks ago. In the usual way with binary search trees, each node in the search tree corresponds to a sub-interval, and you maintain in that node the minimum value on your sub-interval; and this is easy to update as you do rotations. So that works out fine.

The only really tricky question is how you handle the notion of being able to subtract a value from every number in the particular interval. The solution is that you keep *differences* rather than actual values. You essentially arrange for the value of a number in the sequence to be represented as the *sum* of node values on the path from that number to the root of its tree. Again, this is easy to maintain under rotations. But since you're using differences, the augment step simply means that you're going to subtract something from the root — if you decrease the root, then you're decreasing all the sums by the same amount.

This is the general idea. It gets quite a bit messier if you want to deal with trees instead of paths, but the same ideas work. And this gives us a strongly polynomial $O(mn \log n)$ time bound for max-flow.

§10.7 Concluding remarks

You would think that's the end of the story, but that's not even close. We've got this mn time bound, and for a long time, there were strong arguments that you couldn't really do better because of the *flow*

decomposition barrier. The very first thing we showed about max-flow is that you can decompose max-flow into a collection of paths; and then we have all these algorithms that find augmenting paths. If you need a different augmenting path for each unit of flow, then even in a unit capacity graph, you're going to need n different paths, so it's natural you'll need to spend $O(mn)$ time finding these n different paths.

So that seemed like a tight bound. For starters, people tried improving from $O(mn \log n)$; and the Sleator–Tarjan paper actually improved the time bound to $O(mn \log_{m/n} n)$ — so in dense graphs this becomes $O(mn)$, though it still doesn't reach that for sparse graphs.

Eventually Orlin figured out how to get $O(mn)$ with some crazy stuff (which you would never do in practice).

But we're still facing the flow decomposition barrier — a flow requires these m paths, and each of them could be n edges long, so it seems just describing the decomposition would take $O(mn)$ space and time. So we have to get beyond paths.

We got $O(m^{3/2})$, but just for *unit* graphs; Goldberg–Rao after a long time showed how to apply this to a scaling algorithm (this involved various insights about treating certain edges as having 0 length in the admissible graph, and various other things), getting $O(m^{3/2} \log U)$; that got past this barrier.

The next big breakthrough was *push-relabel algorithms*, developed by Goldberg–Tarjan. Instead of thinking all the time about augmenting paths from s to t, they said you should allow things to be out of balance. We stuck with the conservation law that flow in always equals flow out. But they said, let's allow excess to build up at middle nodes. So they described something called a *pre-flow*, which is like a flow (it assigns a flow value to each edge), but only requires that the amount coming in be *no less* than the amount going out — there may be excess going into some other vertices in the graph.

The approach of push-relabel is you find some vertex with excess, and you try to push some of that excess along an edge to a vertex that's closer to the sink. You can imagine you put all these vertices at different heights, and you try to make the flow run downhill to the sink. If you've got a vertex with a downhill neighbor, you send some flow to it; if you have one that *doesn't*, then you increase its heights. These heights correspond to admissible distances. So it's kind of a local search version where instead of building the admissible graph all at once, you're revising your notion of distance based on whether you can get to t, and sending a flow just along one step along a path to t.

This technique is what wins in practice. There are lots of heuristics that never beat the mn time bound, but are really good in practice.

Then in the 2010s, things went really crazy. People started applying continuous optimization and linear algebra to the max-flow algorithm. Lots of breakthroughs actually happened here (at MIT), by Jonathan and Kelner and Alexander Madry. They developed techniques using gradient descent and Newton's method to solve max-flow, which involved formulating the problem as an *electrical flow problem*. One way to get flows in a graph is to think about all the edges as resistors, put voltage at the source, and watch where electricity goes. They showed that by looking at the result and tweaking resistances repeatedly, you can hone in on the max flow.

This led to a whole series of results, culminating into a 2022 paper whose title is 'Max-Flow in almost linear time.' They should have said 'almost max-flow,' because all the Newton methods always have an ε in them. But it's a good dependence on ε, and close to $O(m)$. This paper is 109 pages, so it's not easy and a real tour de force.

But that's where things stand; we have beautiful combinatorial algorithms giving runtimes you can push down to $O(mn)$ or so, and then these push-relabel algorithms which are really good in practice; and then this continuous optimization which gets down to nearly $O(m)$, but is very complicated. These challenge each other — the simplicity of the combinatorial algorithms suggests that maybe you don't need 109 pages, but the near-linear time bounds suggest that maybe you can improve time bounds on the combinatorial algorithms.

§11 October 2, 2024

Last time, we finished up a unit on max-flow, where we developed a reasonably fast strongly polynomial algorithm for max-flow, and used the same ideas to get an even faster weakly polynomial algorithm.

§11.1 Min-cost flow

Today we'll answer a really important question: why didn't we use c for capacities? We'll look at a generalization called the min-cost flow problem, which enriches the max-flow problem with parameters that differentiate between better and worse max-flows.

We start with the flow problem as usual, but we add costs $c(e)$ on edges e. Then we define the cost of a flow f as

$$\mathsf{cost}(f) = \sum c(e)f(e)$$

(so you should think of this as cost per unit flow — there's a difference type of problem where you either buy the edge or not and then you can use as much of it as you want, but here you pay per material shipped along the edge). Note that the costs can be positive or negative (or zero). If you have an edge with negative cost, then when you send stuff on it, you're actually *making* money. (There are many problems where that's an appropriate model.)

Sending f flow on e creates a residual reverse arc, but what should the cost be *on* that residual reverse arc? It should be the negative cost of the original arc — i.e., $-c(e)$ — because the capacity on the residual arc represents removing flow from the forwards arc. And if you remove flow, then you're removing the cost that you paid for sending that flow.

In max-flow, we talked about the net flow formulation, but in min-cost flow you need to keep an eye on the difference between forwards and backwards edges because costs reverse. And you may very well have a use for parallel edges now, which really wasn't required in the max-flow problem — you could combine parallel edges by adding their capacities. But now you might have one small-capacity small-cost edge and another large-capacity large-cost edge to represent it's cheap to send small amounts of flow, but more expensive to send large amounts.

The min-cost flow problem is generally considered what you'd call *min-cost max-flow* — it's taken as a given that you want to send as much flow as possible, and your goal is to minimize the cost of that maximum amount of flow. It's easy to instead solve for the min cost to send a given *amount* of flow (less than the max flow) — if you have an algorithm for min-cost max-flow, you can easily solve this problem. The reduction is to add an edge with no cost and capacity the desired flow, tacked on to either the sink or the source; now the max flow is the flow you want.

Clearly min-cost max-flow is a generalization of max-flow. It can also be seen as a generalization of shortest paths. If we want to send exactly one unit of flow from s to t in a unit-capacity unit-cost graph, the cost of your flow is the sum of edge costs, so the shortest path length is the minimum cost for one unit of flow.

Of course, we still have the flow decomposition into paths and cycles, and you can think about the cost of the min-cost flow as being divided among the paths and cycles (you look at the length of the path and the flow carried along it, and sum over paths and cycles).

§11.2 Min-cost circulation

Once we've added costs, there's another variant of this problem that makes no sense without costs, called the *min cost circulation problem*. This wins for elegance, because there's no source or sink.

Definition 11.1. A circulation is a flow that's balanced everywhere.

You could imagine there are various circulations, and you can ask for the circulation of minimum cost in a graph (rather than the flow of minimum cost).

What can we say about the decomposition of a circulation? It will only have cycles — so a circulation decomposes into cycles.

Note that the min-cost circulation is always nonpositive, because you could just not send any flow — one example of a circulation is the one with no flow everywhere, so you can always achieve a circulation of cost 0. But if there are negative arcs in the graph, then it may be possible to have a negative value for your min-cost circulation.

So the answer is 0 if all the arc costs are nonnegative — if there's no place to make a profit, you can't make a profit. But if you have negative arcs, it's possible you might be able to do better.

You can do reductions between this and min-cost max-flow in various ways. Suppose we want to do a reduction to min-cost flow — suppose we have a min-cost flow algorithm, and someone asks us to solve a min-cost circulation problem. What should we do? We just put a dummy source and sink off to the side, and then ask for the min-cost max-flow. This feels like cheating, but it's not. (If you go back and look at the definition of min-cost max-flow, the amount of flow is still 0, but you're still required to send it at minimum cost; if there is a possibility of getting a negative circulation here, that contributes.)

What about a reduction from min-cost flow? Now I give you a min-cost circulation algorithm, and you want to use it to solve min-cost flow. So now someone gives me a graph with s and t and asks for a min-cost flow, but I only have a circulation algorithm. We can connect t to s — if there's a max-flow, we can send it back to s, and that turns it into a circulation. We put the capacity at ∞, so that it can carry back as much flow as is sent. It's natural to make the cost 0, but this runs into problems. The problem is that maybe this is a graph where it's very expensive to send flow from s to t, and then the circulation algorithm is just going to not send flow. So we instead make the cost $-\infty$. Then whatever flow you send, you're going to get a huge profit for it; so no matter what the costs on the inside are, it's worth it to send flow.

It's not great to work with ∞, since it takes a lot of bits to represent them. So what numbers might we settle as large enough instead of working with actual infinities? Let's suppose we have a maximum capacity of U and a maximum cost of C in the graph (this works even if we're talking about real values). Then we only need to make the capacity of the return arc mU — in the very best/worst case, every single edge connects from s to t and has the maximum capacity, allowing us to send mU flow from s to t.

What about the cost — how negative do I need to make the cost in order to incentivize the delivery of all possible flow? It's enough to take $-nC$ — if we imagine a path decomposition, then it's worth sending flow along the path, because the path is only n edges long and this makes it worth sending flow along the path.

This is fine formally, but it's not great for certain algorithms, because it adds much larger costs. If you talk about weak polynomiality, this doesn't change anything. But if for example we had an algorithm specialized for unit capacity graphs, then this reduction doesn't work anymore. So we want a different reduction with more benefits algorithmically.

Let's suppose we have some max-flow f, but there's some other min-cost flow f^*. We'd like to understand the relationship between these — we're going to sort of develop an algebra of flows. We'll ask, what's the difference? YOu can take that as a general question, but we'll ask it literally — what happens when you subtract one from the other? A flow is just an assignment of numbers on edges, so it's natural to talk about subtracting one from the other. What happens when we do that?

Let's look at the $-f$ term first. If we have a flow from s to t, then $-f$ is a flow from t to s of the same value; it might not be feasible, but it satisfies the balance conditions everywhere. So we can think of it as a flow in the opposite direction.

We're taking a flow from s to t of some value, and attaching it to a flow from t to s of the same value; so we've created a circulation.

Conversely, if we have a flow and we add any circulation to it, we get another flow by the same argument — we're adding something that satisfies the balance constraints everywhere, so we're not changing the net flow into any vertex. Again, this might not be feasible.

But now let's tal about that feasibility.

> **Claim 11.2 —** The flow $f^* - f$ is feasible in the residual graph G_f.

The intuition behind this is that we're subtracting f from every amount of flow, but also from the capacity of every edge, and these balance out.

Proof. First, if $f^*(e) - f(e) \geq 0$, this means there is a flow of this value on edge e. But the capacity of e in G_f is the original capacity minus the flow, i.e., $u(e) - f(e)$. So we have a flow of value $f^*(e) - f(e)$ on an edge of capacity $u(e) - f(e)$. And the fact that f^* was feasible means $f^*(e) \leq u(e)$. So $f^*(e) - f(e) \leq u(e) - f(e)$, which means we're satisfying the capacity constraint.

On the other hand, if $f^*(e) - f(e) \leq 0$, this means we're actually sending flow $f(e) - f^*(e)$ on the *reverse* arc. But in this residual graph, the reverse arc has capacity $f(e)$ (we think about creating a new reverse arc when we need it, rather than adding capacity to an existing arc, because of the different costs). And now we're fine again, because we're sending less than $f(e)$ flow on an edge with capacity $f(e)$.

So either way, we satisfy the capacity constraints. □

This is true of the difference between any two flows; it has nothing to do with min-cost or max-flows.

And that's great — given a starting f, consider *any* feasible circulation q in G_f. By the same argument, $f + q$ is a feasible flow in G. You can think of q as a sort of augmentation, where we're augmenting by a circulation; for the same reason, it produces something which is feasible in the original graph.

Putting these things together, this means the cost of the new flow can be decomposed as $c(f) + c(q)$ (because everything is linear).

This means to get the min-cost flow f^*, what we need to do is add a min-cost circulation from the residual graph to f. To get between any two flows you need to add their difference, which is a circulation; and the change in cost is the cost of the circulation. The circulations you're allowed to add are the ones that are feasible in the residual graph. So the way to do get to the min-cost flow is to add the circulation of minimum cost.

Why is this a better reduction? It didn't introduce any new numbers (or at least, magnitudes) into the flow problem. You simply compute any old max-flow, and now you've transformed min-cost flow into the problem of finding a min-cost circulation in that residual graph.

> **Student Question.** *Could you have added m individual edges instead of one edge of capacity mU?*
>
> **Answer.** Yes, but this creates parallel edges, which is potentially a problem to some bounds.

§11.3 Deciding optimality

As with max-flow, we'll consider a few simpler questions before a full algorithm. Suppose we've got a min-cost flow problem, and someone gives me a particular solution and claims that it is optimal. How do I decide whether it is or not?

Remark 11.3. Nothing's changed about the minimum cut; but now we're no longer saying to send any old flow, we want to find the best possible flow over many. The determination of which flow is the best possible one may have nothing to do with the min-cut — it may be there as a bottleneck for how much flow to send, but you can have choices. So we can't focus just on that.

The thing to observe is that given a flow f, we know (from what we just said) that it's optimal if and only if there is no negative-cost circulation in G_f — put a different way, if the min-cost circulation in the residual graph is not negative. (If we could find a negative circulation, we could add that to the flow and get a cheaper flow. Conversely, if there is no such min-cost circulation, then the above argument shows that there is no cheaper flow — if there was, the difference would give a negative-cost circulation.) And the min-cost circulation is never positive, so you're optimal if and only if $\mathsf{MCC}(G_f) = 0$.

Let's dig in that a bit more. Suppose not, so there's some circulation cost less than 0. I can show you're not optimal by showing you a *circulation*, but those are messy. Just as max-flow suboptimality meant you could find a *path*, we can return to this decomposition idea: here we can decompose the circulation into *cycles*, and the only way for the circulation to have cost less than 0 is if some cycle does.

So if you're not optimal, then there will be a cycle of cost less than 0 in the graph. Conversely, if you have a cycle of cost less than 0, it is a circulation of negative cost, and it can be added to the flow to decrease the cost of the flow, which shows that the flow is not optimal.

So we have a very simple if-and-only-if test — a flow is optimal if and only if there are no negative cycles in the residual graph.

Remark 11.4. Note that the cycle decomposition doesn't have cycles stealing capacity from each other — every cycle in the decomposition is made of edges that are present in the residual graph. (We don't have cycles that aren't there originally but become available once we take one of the other cycles, along a reverse arc.)

Remark 11.5. This optimality condition applies equally to min-cost circulation, for the same reason — your circulation is optimal if and only if there is no negative-cost cycle in the residual graph.

§11.4 Algorithm for min-cost flow

Now we have an optimality condition, which is nice. One reason it's nice is that if you use a LLM to find your min-cost flow, it's nice to be able to check. But another point is that this gives insight into how to develop an *algorithm* for min-cost flow.

We have a max-flow or circulation, but it's not the min-cost one. That means there's a negative-cost cycle. How can we make our flow (or circulation) more optimal? If you're non-optimal, that means there's a negative-cost cycle in the residual graph. So we can just augment on that negative-cost cycle, until it's saturated (i.e., we saturate at least one edge of that cycle). So we find this negative-cost cycle that must be present, and we just send as much flow around it that will fit; and that destroys the min-capacity edge, taking it out of the residual graph, so that is no longer there.

This is known as a *cycle-cancelling algorithm*; it was originally given by Klein in the 1970s. It's basically the analogue of a generic augmenting paths algorithm.

Of course, because you're sending flow along this cycle, you're also creating residual arcs; so you might create new negative-cost cycles not originally in the residual graph (just as sending flow along an augmenting path may create new ones). So this won't trace out a simple greedy path — it may use edges and then un-use them — but the objective function is always increasing.

How many times are we going to need to do this to reach the optimum? The story is very similar to max-flow. If we're talking about integers, the answer is finite; with reals it may not be.

Suppose we have U and C as before (the maximum capacity and maximum cost magnitudes in the graph). Can we bound the number of cycle-cancellings in terms of these values? Similarly to max-flow, we can say it's a negative-cost cycle, which means it's a cycle whose cost is at most -1 (because everything is integral). And its capacity is at least 1. So we improve the cost by 1 on every cycle cancel.

So if we can just bound the minimum possible cost, that'll give us a bound on the number of cycle-cancels we need. The most minimum cost you could possibly achieve is $-mUC$ — which is what you get if every edge has cost $-C$ and you saturate it. And similarly, at the starting point, the cost of the max-flow we start with is at most mUC, for the same reason. So $O(mUC)$ cycle-cancels suffice.

That almost gives us our algorithm, but we do need to be able to find a negative-cost cycle. You might think that's easy — take the negative edges and look for a cycle. That would certainly be a negative-cost cycle. But you may have a graph where all negative-cost cycles involve positive-cost edges as well. (For example, suppose we have an edge $u \to v$ with cost $+1$ and edge $v \to u$ with cost -2.)

We've actually probably solved this in the sense of catching it as an error condition in a different algorithm — namely, shortest paths. What happens if you try to compute shortest paths in a graph with a negative-length cycle? Well, they're not defined. So if we have an algorithm that computes shortest paths, by definition it has to crash if you have a negative cycle (or somehow fail its contract).

So to find a negative cycle, you can just run a shortest paths algorithm and see if it fails. The exact type of failure depends on the algorithm; in many what happens is that the shortest paths never converge (the distance between two vertices keeps going down no matter how many times you run). So you run the algorithm for the requisite amount of time, then run it one more step and see if the distances keep dropping.

We can't use any old shortest paths algorithm; for example, we can't use Djikstra's algorithm because Djikstra's algorithm requires positive edges. (It's using a monotone property; if you have negative edges, you can suddenly discover shorter paths to a vertex you already fixed, because a very negative arc gives you a shorter path.)

But you can use Floyd's algorithm (if we've seen that), or the famous Bellman–Ford algorithm (this is famous because we use it for the Internet). Bellman–Ford runs in $O(mn)$ time, and it's actually an all-pairs shortest paths algorithm (but one crazy thing is once you have edges, you often find yourself computing all pairs just for single-source).

There are also some scaling shortest-paths algorithms that do allow for negative arcs; there's one by Goldberg with a runtime of $O(m\sqrt{n}\log C)$. You can use any of these algorithms — it'll implicitly find negative cost cycles. You may have to think a bit to see where the negative cost cycle actually is, but the algorithm points you in the right direction.

Putting these together, we get $O(mn^2UC)$ from Bellman–Ford, or $O(m^2\sqrt{n}CU\log C)$ from scaling.

All we can say about these algorithms is that they're finite. These are *pseudopolynomial* — they're not polynomial in the input size, because you can fit exponentially large numbers into n bits. But it's a nice starting point.

§11.5 Price functions

It actually turns out cycle cancelling is a very good idea if you do it in more sophisticated ways. But we're next going to go in a different direction.

Cycle cancelling is a good powerful idea, but not one we'll pursue. Instead, we'll take a short vacation into Course 14. We're going to look for a different way to demonstrate the optimality of a given max-flow or circulation.

Recall that a flow is optimal if there are no residual cycles. But we'll think about this differently.

We're trying to solve a min-cost flow algorithm by deciding what to do over the entire graph. But from the perspective of Course 14, that's communism; someone's coming in from the top and saying 'this is where everyone should send their flow and how everyone should make their money.' But this is America, and we are capitalists, so we are going to try to imagine a *market* that will solve the min-cost flow problem for us.

Let's imagine there is a supply of some infinitely precious commodity, such as boba, at your source s; and a demand for this boba at t. And you want to arrange for that boba to make its way from s to t at minimum cost.

You could solve a min-cost flow problem. But an alternative is to offer large payments for boba at t, and give it away at s; and just watch the moneymakers come out and try to make this happen.

What's going to happen is we start to have a market for boba at intermediate vertices — there's going to be a *price* $p(v)$ for boba at v. Where do these prices come from? Well, things have cost because of the cost of getting them to v — these prices emerge from the shipping cost $c(e)$. So if we have an infinite supply of boba at s and there is some cost 10 to send boba from s to v, then the price of boba at v is going to be 10. And this will propagate through the graph — every vertex will have some price at which you can trade boba. And now we want to understand what constraints apply to these prices.

(You can also apply this to the circulation problem — boba is provided everywhere for free, but people can make profit from sending it along certain edges, so you'll have these places where people are sending boba around in circles because they make profit that way; this is maybe not a good model for boba, but it is for other things.)

Question 11.6. When is a set of prices 'stable' or 'correct'?

Suppose we're given two vertices v and w, with prices $p(v)$ and $p(w)$, and we have some cost $c(vw)$ of sending boba from v to w. For example, what if $p(v) = 10$, $p(w) = 17$, and $c(vw) = 2$? This doesn't make sense — then someone at w can open up a new boba shop where they buy it at v for 10 and ship it for 2, and then sell it for 12.

One reason this could happen is if there's no capacity on this edge; then there'd be no way to make money. But if there is capacity, then we'd need the relationship

$$p(v) + c(vw) \geq p(w).$$

You shouldn't be able to make money by buying at v and then shipping and selling at w, and what this says mathematically is that we need $p(w) \leq p(v) + c(vw)$.

Another way this could be stable is if $u_f(vw) = 0$.

And we claim this is the only way you would see a given set of numbers — either the prices satisfy this local cojnstraint $p(w) \leq p(v) + c(vw)$, which we call the *triangle inequality* (because the v looks kind of like a triangle, right?), or there's no shipping capacity.

Taking this triangle inequality thing a bit farther:

Definition 11.7. The reduced cost $c_p(vw)$ (given a set of prices p) is defined as

$$c_p(vw) = c(vw) + p(v) - p(w).$$

If we think about it, this is the cost of buying at v, shipping, and then selling at w.

Definition 11.8. A price function is feasible for a given residual graph G_f if no residual arc has negative reduced cost.

What we're going to see is that a feasible price function shows you that your min-cost flow is optimal.

It's easy to see the opposite direction — if you have a residual arc with negative reduced cost, then this market can operate and change the flow to improve cost.

Claim 11.9 — The prices don't change the min-cost flow — i.e., the min-cost flow is the same under the reduced cost as it was under the original costs.

Proof. The reduced cost of a cycle doesn't change when we throw in prices and look at reduced costs instead of original costs — because all the $+p(v)$'s and $-p(w)$'s cancel each other out. (In other words, the cost of a cycle is unchanged due to telescoping.)

Meanwhile, the reduced cost of *every* s-t path changes by $p(s) - p(t)$, for the same reason.

And now if we think about any flow, the flow can be decomposed into paths and cycles; the total capacity equals the value of the max-flow; and that means every max-flow changes in value by value $\times (p(s) - p(t))$. And this means whatever the min-cost flow was before, it's still the min-cost flow after we insert these prices. □

So prices are very interesting — they're revealing without changing. They don't change the problem you're solving at all, but they make some things obvious — they can demonstrate a circulation is optimal.

Claim 11.10 — A circulation or flow is optimal if and only if there exists a feasible price function in the residual graph.

We've already made the argument that if there is a negative reduced-cost arc (i.e., the price function is not feasible), then the circulation is not optimal. But this is also making the reverse claim.

Proof. First, how do feasible prices in G_f prove that you are optimal? Well, there is a feasible price function, which we call p. What do the reduced costs look like? They're nonnegative on every residual arc. So we have this graph, and looking at its reduced costs, all the arcs are nonnegative.

Before, we said you're optimal if and only if there is no negative cycle in the residual graph. And now we have a graph where there are no negative residual *arcs*. So certainly there is no negative residual cycle.

But wait a minute! This is under the reduced costs. But our optimality condition was talking about negative-cost cycles under the *original* cost. How do we get past that difference? Well, we said the reduction function doesn't change the cost of cycles. So if there's no negative cycles under the reduced costs, there's also no negative-cost cycles under the original costs. And that means we are indeed optimal.

So this is making it clear that prices are just revealing what was already there — they're not changing anything. There may have been some negative arcs, but if you move your costs around, that makes it clear that these were just distractions and there were no negative cycles.

What about the other direction? Suppose we have an optimal f, which means there are no negative cycles in G_f. Our job now is to construct a price function that demonstrates that.

Again, going to our boba example, we've got our graph with boba flowing around, and some residual arcs (maybe some negative ones where someone's paying you for shipment to happen). How do we demonstrate there are no negative cycles? Well, how about the price function that emerges from our boba market?

We're giving the boba away at s; what determines the price elsewhere in the network? If you have a path in the graph from s, you can send boba along the path. So the cheapest way to send boba to that vertex is to take the shortest path.

So we compute shortest paths from s, and suppose we get distances (i.e., shipping costs) $p(v)$ at every vertex v. What can we say about these shipping costs? Well, we want to show that under this price function, all reduced costs are nonnegative. And the reduced cost is

$$c_p(vw) = c(vw) + p(v) - p(w).$$

Can this be negative? No, because if this were negative, then we could get from s to v and then take the edge vw, and this would give a shorter path than what we think is the shortest path to w. So this must be nonnegative, by the triangle inequality for shortest paths.

So if there is no negative cycle, a shortest paths computation will construct prices that are a feasible price function, and demonstrate that there is no negative cycle. Essentially what we're doing is we've got a negative arc, and by introducing prices we sort of push that negativity elsehwere. We're adding $p(v)$ and subtracting $p(w)$; so if we have an arc of -10 and put 10 at $p(v)$, the reduced cost here turns to 0. But the other effect of this is that now an incoming arc is going to lose that amount. And if this arc was previously very positive, then we can move the negative quantity here and have it be cancelled out by the positive quantity earlier. So these prices are just shifting costs around so that the apparently negative parts vanish and are swallowed by the positive parts.

There's one problem — what if there is a vertex that's not reachable from s? In that case, there is no shortest path from s to that vertex. This is only a technical problem, because if you can't reach the vertex from s, then you can't send flow and it's irrelevant to the rest of your calculations. But it's nice to define your price function everywhere. To deal with this, what you do is you add a new source s', with cost-0 edges to *every* v. This does not create any negative cycles, because there is no way to get to s' (so it can't participate). And now you run shortest paths from s' instead — now there's a path to every vertex, so you have defined shortest paths. □

So far, this is just a characterization — there's nothing algorithmic here. It doesn't matter which shortest path algorithm you use. But the point is it'll only produce these prices if there are no negative cycles — if there are, then shortest paths aren't defined (the algorithm won't converge).

So here we have a nice *local* characterization of optimality, and one which is much easier to test — if someone gives you a flow and says 'I'm optimal,' you need to do some calculation to find whether there is a negative-cost cycle. But if they also give you a price function, all you need to do is check that every residual arc has nonnegative cost; you don't have to run any more complicated algorithm.

So we say these prices form an easily checkable *certificate* of optimality. That's not a formal term because you still have to run an algorithm (to check that edges have negative reduced cost); but that's a much simpler process.

So far we only have this characterization. But just as we have the cycle characterization to lead us to a cycle-cancelling algorithm, we can also use prices to get a price-calculation algorithm.

> **Remark 11.11.** There is no class on Friday.

§12 October 7, 2024

Today we'll finish up min-cost flow and probably start on linear programming.

§12.1 Review

Last time, we got introduced to the notion of price functions, and solving min-cost flow through capitalism. We assign a price $p(v)$ to every vertex, and we define the *reduced cost* of an edge as

$$c_p(vw) = c(vw) + p(v) - p(w),$$

which represents the cost of purchasing some quantity of material at v, shipping over to w, and then selling it and recovering the price at w.

We observed that reduced prices don't change cycle costs, and they change all s-t path costs by the same amount, which means that shortest paths stay shortest, and more generally, min-cost flows stay min-cost flows.

We then declared a price function to be *feasible* for a given flow if all the reduced costs are positive (in the residual graph of that flow). If all the reduced costs are nonnegative on residual arcs, this means there is no negative reduced cost cycle, which means there is no negative real cost cycle (by the above); this implies that our flow is a min-cost flow. So these feasible price functions provide a kind of certification that whatever flow you're looking at is indeed the min-cost flow. And it's a much simpler certification to check — you're given the price function and you compute all the reduced costs, check that they're nonnegative, and you're done. You don't have to do anything like calculating if there are negative-cost cycles; so it's really easy to verify.

But today we're going to talk about how price functions are also great for computing min-cost flows. Right now, our current state of the art is that we have a cycle-cancelling algorithm, which is pseudopolynomial in U and C, but we don't yet have a polynomial-time algorithm for this problem. That's the first thing we're going to develop today.

§12.2 A greedy approach

We'll start with a pretty natural idea, which is to be *greedy* in computing a min-cost flow.

We'll change perspective — we've been talking about circulations a lot, but for now we'll talk about the opposite. We'll assume we have a graph with no negative cycles. In that sense, our flow is already min-cost when we start with no flow at all. But we're looking for a min-cost *max*-flow in this graph.

Now, given all the work we've been doing to understand the min-cost flow problem, if we want to augment a flow (in the augmenting paths perspective from max-flow algorithms), what's the natural augmentation to do if we're pursuing a flow that in the end will be min-cost? If we're going to look for a min-cost flow, it's natural to try to augmnet along the min-cost path (the shortest path under the edge costs).

So we're back to considering a shortest augmenting paths algorithm. But critically, our change is that now we don't talk about distance in terms of number of edges (the way we did in max-flow). Instead we talk about the shortest augmenting path under the edge costs c. In our homework, we'll see that these things are not by any means the same — you can't use lengths and end up with a min-cost flow. But if you use costs, it turns out you do — that simply repeating augmentations under the min-cost paths gives you a min-cost flow.

Why? We'll actually give two separate proofs, which will both show that augmenting along a shortest path never creates negative-cost cycles. So if we do augmentations that don't create negative-cost cycles, then when we finish (and have a max-flow), we'll have a max-flow and we won't have any negative-cost cycles in the residual graph. And that means it's a min-cost max-flow.

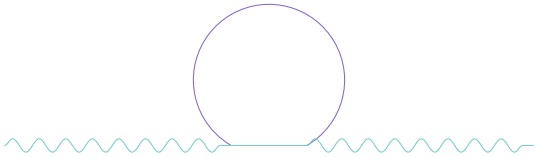
> **Claim 12.1 —** Augmenting along a min-cost path doesn't create negative-cost cycles.

Proof. Suppose that we do create a negative-cost cycle. Let's think about how that could happen. There wasn't a cycle of negative cost in the graph, we did an augmentation, and now there is a negative cycle. How could that have happened? The only way to create a negative-cost cycle is if the augmentation introduced new edges that participate in that cycle. So this negative-cost cycle can't be made entirely of edges that were present before we did the shortest augmenting path — if the edges were already there, then we would have had a negative-cost cycle even before.

So we have a negative cost cycle, and importantly, at least one edge in this negative-cost cycle must be a new edge. It might not be a negative-cost edge — it might be a positive-cost edge, but one completing a cycle that in total has negative cost.

And how could we have created a new edge in the residual graph? That means the augmenting path used that edge in the reverse direction.

But now from this picture, we didn't use the shortest augmenting path. We thought that this was the shortest augmenting path, but it's shorter to instead go through the cycle. Our cycle is a negative-cost cycle, so if we went along that edge as before, then did a u-turn (this is Boston) and went around this cycle and then to t, then we've added to the path the value of this entire cycle. (Equivalently, we're taking the rest of the cycle instead of this edge.) So the cost of this different path is less than the one we had before.



In other words, the shortest augmenting path traversed an edge of the new cycle in the opposite direction; this critically means the cycle is attached to the augmenting path that we found. And that means the augmenting path plus the cycle is a cheaper augmenting path, which is a contradiction to the idea that we were using a shortest augmenting path. □

Student Question. *What happens if the shortest augmenting path touches the cycle twice?*

Answer. It gets more complicated, but the point is that if you take the sum of the cycle and path, it's still a path — there may be some cancellation of various edges and things, but adding the cycle either keeps you a path or breaks off a piece (which only helps you, since that piece that breaks off can't have been a negative-cost cycle).

Proof 2. Recall that any price function changes the *values* of paths, but not their relative order — the shortest paths are still the shortest paths.

Given that there are no negative cycles at the start, we can compute prices as distances from s, exactly as we did last time — where $p(v)$ is the distance from s to v. We saw that this is feasible by the triangle inequality — i.e., $c_p(vw) = c(vw) + p(v) - p(w) \geq 0$.

But the edges on shortest paths from s all have reduced cost 0, just by definition — $p(v)$ and $p(w)$ are the distances to v and w, and $p(w) = p(v) + c(vw)$ (this is what it means for vw to be on the shortest path).

So all the edges on shortest paths from s have reduced cost 0; this of course includes the shortest augmenting path (the shortest path from s to t). So we're augmenting along a path whose edges all cost 0. Then the costs of the residual arcs we create are also all 0.

But if the new residual arcs have cost 0, then there are no new negative-cost residual arcs! We started out with no negative-cost arcs, and so we still have none (under the same price function) — i.e., the price function p is still feasible after the augmentation.

(Here we're working with *reduced* costs, not actual costs — all the *reduced* costs are nonnegative, and they stay nonnegative under augmentation.)

The price function might no longer be the shortest-paths function — we destroyed some edges — but that doesn't matter. □

This is nice because it lets us think about the problem entirely locally.

§12.3 A shortest augmenting paths algorithm

Now we have two proofs that shortest augmenting paths never create negative-cost cycles, and that lets us think about shortest augmenting paths algorithms for min-cost flow.

If we wanted to do this directly, it wouldn't get us too much. If we did this in general graphs, assuming there are no negative-cost cycles to start with, we could use Bellman–Ford to perform the shortest paths computation. If the graph is integral, then we know the max-flow is at most mU, and we're going to be finding one unit per shortest augmenting path; so we only need mU shortest augmenting paths. And each costs mn, so we have an $O(m^2nU)$ algorithm. This is not much better than cycle-cancelling, though it is better in one interesting way — there's no dependence on the costs. For all we care, the costs could even be real numbers, and this algorithm would still terminate. So that's kind of nice, but it's not much.

So let's try to be smarter about it.

We can use reduced costs to speed this up. If we know there are no negative-cost cycles, then we can compute shortest paths from s, and these shortest paths give us nonnegative reduced costs. Now we do a shortest augmenting path; and this *still* yields nonnegative reduced costs (we just proved that). And if the reduced costs are still nonnegative, then for the next round we don't need to use Bellman–Ford to find the shortest augmenting path — we can use Djikstra instead, for the next (and future) shortest augmenting path rounds, which involve computing distances and a new price function and then augmenting along some zero-cost edges.

So feasible prices aren't just useful as a certification trick, they're also useful for the algorithm — they keep us from being distracted by negative edges, so that we can use a more efficient shortest-paths algorithm.

Note that when we do our shortest augmenting path, we don't create negative arcs, but we might change the distances; so we really do need to recompute the price function in every round. But nonnegativity lets us use Djikstra for that. And this improves our runtime from $O(m^2nU)$ to $\widetilde{O}(m^2U)$ (there's some $\log n$ from heaps).

> **Student Question.** *What happens if you start with negative-cost cycles?*
>
> **Answer.** Then the Bellman–Ford algorithm doesn't give you shortest paths, because they don't exist; so this algorithm just doesn't work. That's why we're assuming for now that there are no negative-cost cycles. We will deal with those later (or maybe now).

> **Student Question.** *Why can we use Djikstra?*
>
> **Answer.** We need Bellman–Ford the first time to figure out our prices. Those prices yield edges that have nonnegative reduced costs. Then we do shortest augmenting paths and need to compute a new price function, but we do that using the *reduced costs* (under the old price function), which are all nonnegative. So at every iteration, I'm working off the reduced costs from the *previous iteration*, together with some new zero-cost edges.

We can actually refine this to $\widetilde{O}(mf)$ (one mU in the previous bound comes from the bound on the max-flow). (Technically there's also an $O(mn)$ from Bellman–Ford, but typically mf dominates.)

§12.4 Dealing with negative cycles

We've got an algorithm with no negative cycles, but what if there are negative cycles? We'll just brute-force them away. We switch perspective back to min-cost circulation — we start by finding any max-flow, so that we want a min-cost circulation in the residual graph.

Then we get rid of negative-cost cycles in the most brute-force way possible — we simply saturate all the negative arcs with flow. What does this mean? We go back to our boba economy, and I come in and see,

there's a negative-cost arc here with capacity 10; I insist you ship 10 units of boba from one side to the other, using up all of its capacity. Suddenly, boom! there's no negative arcs.

But now we've created a different problem — now we don't have a flow. What we have are *excesses* and *deficits* at vertices. Some of the vertices we're shipping large amounts into them, and they have a large pile of boba now; for other vertices, we're taking boba they don't have. But we've solved our problem of having negative-cost cycles, because we don't have negative arcs at all; all we need to do is solve our problem of having too much boba at some vertices and not enough at others.

We have stuff in one place and want it to be in another place. And as on the homework, we can do this by a flow algorithm — we run a flow algorithm to ship the excess to the deficits. Is it possible to do so — what if there's no solution? For one thing, we know the excess and deficit are equal. But how do we know there's capacity?

Well, there is a feasible solution that uses all the residual arcs I just created. I said to send flow along this arc, and that created a residual arc in the opposite direction. This is a positive-cost residual arc, but I can still use it — so there is a feasible solution to this flow problem, and therefore there exists a min-cost feasible solution, i.e., a min-cost flow.

So we've saturated all the negative-cost arcs by creating some excesses and deficits, and we know there's a feasible flow for returning the excesses to deficits, and therefore there's a min-cost flow for that.

> **Claim 12.2 —** The original saturation plus the min-cost flow returning excesses to deficits is a min-cost circulation of the original graph.

This is the same argument from last time — we saw that when you find any old flow and add a min-cost circulation of the residual graph, you get a min-cost flow. That's what we're doing here — or we're kind of doing the opposite (creating a min-cost circulation by adding a min-cost flow to a flow in the opposite direction). (The point is that anything completing your flow into a circulation is a flow in the opposite direction, so you want to take the cheapest one.)

So we have now reduced the problem of computing a min-cost circulation in a graph with negative cycles, to the problem of computing a min-cost *flow* in a graph *without* negative cycles (actually in this reduction, we have no negative arcs). And conveniently, we've just developed an algorithm for that. (In fact, here we don't need an initial stage of Bellman–Ford, because the graph we start with already has no negative arcs; so we can just jump in with Djikstra.)

So we can solve min-cost circulation in $\widetilde{O}(mf')$. What is f' here, though (the flow we're attempting to ship)? It's the total capacity of all the negative arcs, which is bounded by mU. So this is still $\widetilde{O}(m^2U)$ time.

So now we have a general algorithm for min-cost circulation that makes no assumptions about negative arcs or cycles — it takes a graph with whatever negative arcs you want, and finds a min-cost circulation.

> **Remark 12.3.** This solves a min-cost circulation problem by turning it into a min-cost flow problem with no negative arcs. Now if you start with a min-cost *flow* problem (with negative arcs), you reduce min-cost flow to min-cost circulation (by finding any old max-flow — to find a min-cost flow, you find any old max-flow and then find a min-cost circulation in the residual graph, and we just showed how to do that). It's a bit perverse — you start with min-cost flow, turn it into min-cost circulation, and then go back to min-cost flow. But what's important is what you gain — the first reduction gets rid of s and t to get a supply and demand problem.

§12.5 Scaling min-cost circulation

We now have an algorithm that works wonderfully as long as U is small. What if U is big? Maybe we have a method that allows us to transform an algorithm that works for small U into an algorithm that works for large U?

The answer is scaling; this was discovered by Edmonds–Karp 1972. (Edmonds was one of the founders of optimization, and studied a lot about matroids and matchings. Karp is David's advisor's advisor; he published a very important paper immediately after the invention of NP-completeness, where Cook and Levin showed that there is one NP-complete problem — Karp published a seminal paper showing how you could use reductions to show a bunch of others were too. So he developed the idea of connecting all these problems together to show they're equally hard, among other things.)

We're going to use capacity scaling — so we start with our capacities rounded down to 0, and then we roll in one bit of capacity at a time. Suppose we finished the phase with an optimal circulation and a feasible price function p. In each scaling step, we double the capacities and our current circulation (it's easier to talk about scaling for min-cost circulation, though we've seen you can go back and forth). Then we add 1 to certain capacities. It's this adding of 1 to certain capacities that make our solution possibly non-optimal.

In what sense might the current circulation be non-optimal? We've done this scaling step and created new capacity, which might be on edges that had negative cost. So that's the problem — all of a sudden we're not sure our solution is optimal, because there might be some negative-cost edges in the graph.

So how do we fix this? Again negative arcs are not necessarily a problem, but they might create negative-cost cycles, which are a problem. So how do we fix this? Well, we use the algorithm we just defined — we have a graph and some negative arcs, so we just saturate all of them and use the nonnegative arc min-cost flow algorithm to fix the excesses and deficits.

Why is this better than just trying to solve the whole graph at once? The only arcs that are negative are the arcs we just created, and those all have capacity 1. So this means the total excess or deficit created is at most m, and therefore our runtime of $\widetilde{O}(mf)$ is just $\widetilde{O}(m^2)$ — we need m shortest augmenting paths and each takes $\widetilde{O}(m)$ time, and this fixes the scaled capacities.

And we have to do this scaling phase $\log U$ times, so now we have an algorithm with a runtime of $\widetilde{O}(m^2 \log U)$ for min-cost circulation. And that's a polynomial-time algorithm!

Student Question. *What if the extra capacities increased the flow?*

Answer. This is for min-cost circulation, so there is no flow to increase. (We've seen that if you can solve min-cost circulation, you can use it to solve min-cost flow, by first finding a max-flow and then using this to fix it.)

Remark 12.4. Min-cost circulation doesn't have the same relation to cuts that max-flow does — you're trying to saturate the negative cycles instead of cuts. We talked about how min-cut is the 'dual' to max-flow; there is also a dual to min-cost flow, but it's much less obvious (and will be on the homework).

Note that this has log dependence on U, but it has no dependence on the cost — it's polynomial time even if the costs are real numbers. On the homework, we do a cost-scaling algorithm, which has log dependence on C but allows real capacities. You can wonder if there's a strongly polynomial algorithm; the answer is yes, but we're not going to discuss the details.

§12.6 Complementary slackness

(This name may not make a lot of sense now, but it foreshadows linear programming, where it will make a lot of sense.)

Let's think about our prices and reduced costs again, and let's think about the independent truckers who are traversing one edge, buying boba at the tail and selling it at the head.

Let's suppose that the reduced cost of an edge is positive. What are shippers going to decide to do? This means $c(vw) + p(v) - p(w) > 0$. That means they're making a loss, so there will be no shipments along this

edge. We've got this price function and reduced costs, and if there is an edge of the original graph with positive reduced cost, then no one will use it.

Suppose that the reduced cost is negative. Then the shipper is incentivized to use it, as much as they can — so the edge will be saturated (someone can make money by shipping on it, so they will — this is in our model of having infinite amounts of boba that can be picked up and moved).

Meanwhile, if the reduced cost is 0, then who knows? If the driver enjoys the scenery, maybe they'll pick some stuff up and sell it, but it won't make them a profit (or loss).

Definition 12.5. A flow with these characteristics is said to have complementary slackness to the price function.

We're looking at the flow and price function on the original graph, and there's this relationship between the flow and the price function.

And we claim this is another optimality condition.

Claim 12.6 — Complementary slackness is equivalent to optimality of a max-flow.

Proof. This is just a matter of following through the definitions — we know your max-flow is optimal if there are no negative-reduced-cost *residual* arcs. But this condition says that a negative-reduced cost arc in the original is saturated, so it's not in the residual graph. Meanwhile, if there's a positive-reduced cost arc in the original graph, then its reverse arc would be negative, but we don't *use* that arc.

So if you have complementary slackness for some price function, then that's a feasible price function, and you have a min-cost flow. Conversely, if you have a min-cost flow, it will exhibit complementary slackness with the distance function in the residual graph (for the same reason — the residual graph has no negative-cost residual arcs, and that means all the negative-cost arcs in the original graph are saturated (or else they'd have residual capacity) and the positive-cost arcs are unused (otherwise you'd be creating residual capacity in the opposite direction)). □

So complementary slackness says the same thing as before, but in terms of the original rather than residual graph.

But there's one other very nice thing about complementary slackness that kind of unpacks what's going on. Suppose we're given a feasible price function for a min-cost circulation problem — we don't know what the circulation is, we're just given the feasible price function. What does complementary slackness tell us about the optimal circulation? Well, let's look at the reduced costs. Complementary slackness says that you don't use any arcs with $c_p > 0$ — you're not allowed to send flow on them. So we can just delete those arcs. Complementary slackness also says that you should saturate all the negative-cost arcs, so we do so. Well, saturating those arcs creates excesses and deficits. What does the min-cost circulation have to do? Well, it has to route the excesses back to the deficits. What arcs is it allowed to use to do so? Only the arcs with reduced cost 0.

And what algorithm do we need to solve the problem of routing excesses to deficits on zero-cost arcs? It doesn't matter which arcs we use — they're all 0 cost. So we can use a normal max-flow algorithm (for supplies and demands).

So if someone gives me a feasible price function, I don't have to worry about the costs anymore — it becomes a normal max-flow problem. This is sort of the statement 'min-cost flow is max-flow plus shortest paths.' If you have the flow then you can compute the prices, but if you have the prices you can also compute the flow just using ordinary max-flow.

We'll see a lot more about complementary slackness when we do linear programming.

§12.7 Conclusion

Min-cost flow is a pretty important problem — lots of real-world problems can be formulated as min-cost flows (there's a few on the problem set; it's fun when the admissions office says 'we have this really complicated problem, can you solve it for us?' and the answer is 'yes, it's just min-cost flow').

Strongly polynomial algorithms do exist. The first one was developed by Tardos (another very important name in optimization) in 1985. it used a concept that we haven't looked at, called *minimum mean-cost cycle*. We did shortest augmenting paths, and you could also imagine looking for the most negative cycle in a cycle-cancelling algorithm. But it turns out you get better results if you look for a cycle that has the minimum *average cost per edge*. So you might take a cycle that is not actually the minimum cost if it uses fewer edges, so that the cost per edge is lower. (Presumably a lot of these edges are negative-cost.) Conversely, if you're working in a graph with all positive edges, the min-cost cycle might be a little thing that doesn't do very much in the graph; while there might be a very long cycle that has minimum mean cost because it's being divided by many edges.

The other idea she uses is sort of a scaling idea, but it turns out you can sort of use scaling ideas in strongly polynomial algorithms. So this is a scaling-like algorithm, but at various times it says 'on this edge, the extra scaling is not going to change anything' — so it's going to 'fix' certain arcs as empty or saturated after some strongly polynomial amount of time. This relates to the complementary slackness idea. Imagine I'm doing all this scaling and I've got these reduced costs, and I see an edge with really large positive cost. That cost might change a little bit as I bringin more bits, but it's never going to go negative, and that means it has to be empty in the final solution. So I can just fix that edge as empty and delete it. Similarly, if I have an edge that's very negative, it's never going to become positive, and that means I can fix it as saturated and stop paying attention. Tardos showed that by running minimum mean cycles, you can look at certain edges and say I know they'll ultimately be positive or negative reduced cost at their end, so I can fix their value — this may create excess or deficit, but you don't have to worry about them anymore. You keep removing more and more edges, and eventually you get your max-flow.

The runtime of this algorithm ends up something like $O(m^2)$. This is still a little bit unsatisfactory, because for max-flow we got $O(mn)$ even before the recent breakthroughs on continuous optimization.

You can go even farther with scaling. There's an algorithm that uses *double* scaling (on both cost and capacity); this is by Orlin, and has a runtime of $O(mn \log C \log \log U)$. So it's weakly polynomial depending on both capacities and costs, but when those are not crazy, it approaches the mn runtimes of our best max-flow algorithms.

Next class we'll begin linear programming, which subsumes both max-flow and min-cost flow and basically all optimization problems; it's a sort of sledgehammer.

§13 October 9, 2024

This week is about linear programming, the biggest sledgehammer in combinatorial optimization.

§13.1 Linear programming

We've seen the max-flow problem, where we want to maximize $\sum_v f(s,v) - f(v,s)$ under various constraints: $0 \le f(v,w) \le u(v,w)$, and various balance constraints $\sum_w f(v,w) - \sum_w f(w,v) = 0$ (unless $v = s,t$).

Written this way, it's a problem of optimizing a linear objective function over certain variables, subject to various linear inequality and equality constraints. That's what linear programming is about — take *any* problem with a bunch of variables, a bunch of linear equalities and equalities, and a linear objective function you want to optimize. How do you do that?

You can do the same for min-cost flow, and in fact, also for shortest paths — all the combinatorial optimization problems we've looked at in this class are special cases of linear programming.

In the general form of linear programming, you have a collection of variables, and *constraints*, which are linear equalities and inequalities involving these variables. We're working in the general case, so we'll often use vector notation, with $\vec{x}$ representing all the variables we want to work with; a single constraint can generally be seen as a dot product of the variable with some constants (that creates a linear function of the variables).

Importantly, all inequalities are $\leq$ and $\geq$, not $>$ and $<$.

Definition 13.1. We say $\vec{x}$ is feasible if it satisfies all the constraints.

Definition 13.2. We call the variables together with the constraints a linear program.

Definition 13.3. We say the linear program is feasible if there exists some feasible point $\vec{x}$ for it.

Definition 13.4. We say $\vec{x}$ is optimal if it optimizes a certain linear *objective function* over all feasible points.

(This is just some basic vocabulary.)

Some linear programs are infeasible — they don't have any feasible points at all. Of linear programs that *are* feasible, only some have optima: you may also have linear programs that are unbounded, if there are points of arbitrarily good objective values.

Example 13.5

Consider a linear program with no constraints, and objective function x: this is an unbounded linear program, because we can make x as big as we like.

It should be obvious that every linear program is either infeasible, has an optimum, or is unbounded. (It's always one of these three possibilities.) But which one is always a very interesting question. The proof follows form the compactness of $\mathbb{R}$ and the fact that our inequalities define a closed set. (This is why it's important we use $\geq$ and $\leq$ — if you used strict inequalities, you'd have problems where there might be a limit you can't actually get to.)

§13.2 Some notation

§13.2.1 Canonical form

We'll now introduce some standard notation we'll use for linear programming. A common form is the *canonical form*:

Problem 13.6 (Canonical form)

Maximize $c^\intercal x$ subject to the constraint $Ax \leq b$.

Here x is your vector of variables and c a vector of constants, so $c^\intercal x$ is your linear objective function. Meanwhile in $Ax \leq b$, the idea is that on the left-hand side you have a matrix where each row defines a linear function on x, and the right-hand side is a vector of bounds on those function — so the number of rows of A is the same as that of b. (By $\leq$, we mean component-wise or row-wise.)

Definition 13.7. We call A the constraint matrix, and we call c the objective.

What's important is that *any* linear program can be transformed into canonical form. So if I give you an algorithm for solving linear programs in canonical form, then it's actually a general purpose algorithm. Let's verify that transformation:

- Suppose we have a *minimization* linear program instead of a maximization linear program. How do we flip this? We can simply negate the objective function — we have $\min c^\intercal x = -\max -c^\intercal x$.
- Suppose someone gives me a linear constraint $a_1x_1 + a_2x_2 + \cdots + k \leq b_1x_1 + b_2x_2 + \cdots$, with variables all over the place. How do we transform this so that the constraint is in canonical form? We can just move all the x_i's to one side and the constants to the other, and that fixes the placement of the variables.
- What if we have constraints of the form $ax \geq b$? We can fix this by negating both sides and writing $(-a)x \leq -b$.
- What if we have constraints of the form $ax = b$? Then we can turn this into two inequalities $ax \geq b$ and $ax \leq b$ (and adjust one by negation).

So that shows it's easy to transform any linear program into canonical form.

§13.2.2 Standard form

Canonical form is very elegant, but it's actually not the best form for developing algorithms. So we also have another form.

Problem 13.8 (Standard form)
Minimize cx subject to $Ax = b$ and $x \geq 0$.

(We're sometimes going to omit the transposes.)

So here we have a bunch of *equality* constraints, and a nonnegativity conditino.

Again, every LP can be transformed to standard form. Now we don't have to consider so many cases — we can assume we have a LP in canonical form, and we just need to transform *that* into standard form. But how can we transform $Ax \geq b$ into this form?

The idea is that we introduce 'slack variables' s_i, with one for each row of the original. We'll transform $a_ix \leq b_i$ into $a_ix + s_i = b_i$, and $s_i \geq 0$. And this achieves the same thing — if $a_ix \leq b_i$ then it's less by some nonnegative amount, and s_i can be that amount.

That gives us equality constraints, and introduces variables that are required to be nonnegative, as they should be. But what about the x's — we can't just throw in a bunch of $x \geq 0$ constraints, that would change our program. So how can we work with nonnegative variables but still have as much freedom as unconstrained ones? What we do is write $x = x^+ - x^-$, where $x^+, x^- \geq 0$. So if we want x to be a positive variable we could set x^+ to that value and x^- to 0; if we want x to have a negative value then we set x^- to the negation of that value and x^+ to 0. So this is just as expressive as the original x was, but now all our variables can be constrained to be nonnegative.

Now we've shown that we have two different forms of LPs, and either is fully general. It will turn out, as mentioned, that canonical form is often very nice for thought and theory and intuition, but most algorithms like standard form, for reasons that will become clear later.

There's also a very interesting relationship between canonical form and standard form that we will see on Friday.

§13.3 An overview

This is the formulation of our problem; we'll spend more than a week figuring out how to go about solving it. We're going to take the same sequence of steps towards a solution as for max-flow and min-cost flow.

> **Question 13.9.** What does an answer look like?

There's actually a fundamental problem for linear programming — can we write the answer down? Maybe the answer is a bunch of real numbers that don't fit in our computer; that's something to worry about. This is kind of equivalent to asking, is it rational and does it not have too many bits?

The next step:

> **Question 13.10.** Can we verify an optimum?

> **Question 13.11.** Can we find a feasible point (that just satisfies the constraints)?

> **Question 13.12.** Can we prove that there is no feasible point?

Proving an LP is feasible seems relatively straightforward (I just give you a point). But how in the world would you show that a LP is *not* feasible?

Then we'll finally ask:

> **Question 13.13.** Can feasible solutions, optimal solutions, and proofs of infeasibility be found efficiently?

§13.4 The linear equality case

So that's our process. We'll demonstrate this at a baby level by first thinking about systems of linear *equalities* — so for a while we're just going to talk about the good old system $Ax = b$. On the surface this actually doesn't seem that different from linear programming — for linear programming you just need to toss in the nonnegativity constraints. But it turns out that makes things incredibly more difficult. Still, even here there's some interesting stuff to think about.

> **Question 13.14.** When does $Ax = b$ have a solution?

Let's start with the easy case where A is a square matrix, and we've got a linear system $Ax = b$. What's the easy way to demonstrate there is a solution? Well, you can give one. If someone gives you a solution, it's very easy to verify it's correct — you just plug it into the system of linear equalities and make sure they're correct.

> **Question 13.15.** Can you *construct* a solution, if it exists?

Yes, you can do this by Gaussian elimination (which David is assuming we learned in preschool). It's a process where you start subtracting one row from another in order to turn the matrix into triangular form, and then you get rid of a bunch of other stuff and just have a diagonal, and that lets you read off the solution.

We have a bunch of equivalent statements regarding this linear system:

Fact 13.16 — The following are equivalent:

- The matrix A is invertible.
- $\det(A) \neq 0$
- The rows of A are linearly independent.
- The columns of A are linearly independent.
- $Ax = b$ has a unique solution for every b.

(We probably learned this in 18.0X — if the determinant is nonzero then you can construct the inverse of A, and then you can set $x = A^{-1}b$.)

§13.4.1 Sizes of numbers

This all worked when we got homework problems that were carefully constructed to be nice and have nice answers. But do we know that the answer is always nice — that we can write it down in a reasonable amount of space?

We've already talked at length about how an integer n needs $\mathsf{size}(n) = \log n$ bits to represent it. Similarly, a rational number p/q can be represented using $\mathsf{size}(p) + \mathsf{size}(q)$ bits (writing the numerator and denominator separately).

We'll be doing Gaussian elimination and inversion and stuff, so we need to think about products and sums. If we add two numbers, how big could their relative sizes be? We can say

$$\mathsf{size}(x + y) \leq 1 + \max\{\mathsf{size}(x), \mathsf{size}(y)\}.$$

Meanwhile, we have

$$\mathsf{size}(xy) \leq \mathsf{size}(x) + \mathsf{size}(y).$$

(This is basically because logs add.) If we have a n-vector, then its size is n times the size of its entries; similarly, the size of an $n \times m$ matrix is nm times hte size of its numbers.

We've got these notions of sizes, and what we're worried about now is algorithms whose runtime will be polynomial in the size of our input — that involves both the dimensions of the matrix, and also the bits used to represent the numbers in the matrix.

So let's think about the issue of writing down the answer. We could approach this by looking at Gaussian elimination, but we'll actually do it by looking at $\mathsf{size}(A^{-1})$. To understand this, we can go to the definition of A^{-1} as

$$\frac{1}{\det(A)} \cdot (\text{cofactor matrix}),$$

where the cofactor matrix is also a bunch of determinants. So the real question we need to ask is about the size of $\det(A)$. How big can the numbers in $\det(A)$ be?

First, is it going to be rational? You can write

$$\det(A) = \sum_{\pi \in S_n} \operatorname{sgn}(\pi) \cdot \prod_{i=1}^{n} a_{i\pi(i)}$$

(where we're picking one entry out of each row of A, and multiplying all those entries together). (You can think of these products as matchings between the rows and columns; there are deep connections between determinants and matchings, which we'll see if we take randomized algorithms.)

On the inside we're taking a product of n numbers, so we'll end up with something whose size is roughly $n \cdot \mathsf{size}(a_{ij})$. And then we're summing up these numbers over all permutations. And there's $n!$ different

permutations, which means that when we take this sum, we're not doing more than multiplying this quantity by $n!$. So if we put these together, we get $\mathsf{size} = n \cdot \mathsf{size}(a_{ij}) + \mathsf{size}(n!)$. And how many bits do you need to represent $n!$? We have $\log n! \approx n \log n$ (up to constants).

So this is somewhat promising — whatever the size of the numbers you start with, taking a determinant may increase those numbers by a factor of n or an additive $n \log n$, but not worse than that. So the output numbers are still polynomial in size compared to the input.

So we are not immediately faced with a massive obstacle; we know that we can in fact write down the answer.

You can also show this by looking at how the numbers grow as you perform Gaussian elimination, which also shows that the numbers don't get too big.

And since the determinants are polynomial-sized, each entry in the inverse matrix is a ratio of two determinants, so the whole inverse is also polynomial-sized.

> **Student Question.** *Don't we have to worry about the denominators, which need to get multiplied?*
>
> **Answer.** One thing you can do is scale things up and then divide out by the least common denominator, which is still polynomial-sized.

(In practice you're going to be working with floating points, and then you have to worry about loss of precision; but this is a theory class, so we'll focus on the case where you have exact numbers and we worry about having enough bits to represent them.)

We focused on square matrices, but all of this carries over — you can still use Gaussian elimination to find the solution. And in Gaussian elimination you're taking two rows and using one to cancel the other, meaning you're multiplying numbers in one row by numbers in another, and then you're adding these. And you do this n times, so even if the matrix is not square, you still end up multiplying the size by roughly n.

§13.4.2 Showing infeasibility

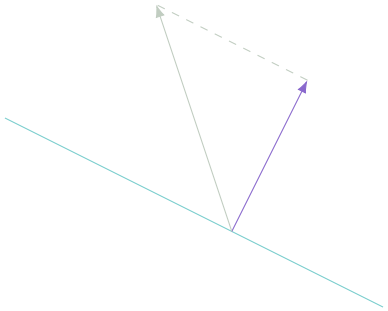
> **Question 13.17.** Can we show there's no solution to a linear system?

One approach is to say, there's an algorithm that finds a solution if there is one; if I run this and it doesn't find a solution, then there isn't one. But can you just show them something that's easy to verify directly, instead of having to verify the execution of an entire algorithm? That's what we're going to do now.

The *existence* of a solution to $Ax = b$ has an easy witness, namely x. We don't want to have to use failure of algorithms to prove the opposite. So: if we're trying to solve $Ax = b$, this means b is in the span of the columns of A — x defines a linear combination of the columns of A, and that produces b. If we want to show there is no solution, what we're trying to show is that b is not spanned by the columns of A.

Now, the set of points $\{Ax\}$ is a linear subspace, and we want to show it doesn't contain b. So let's look at a very simple 2D case of that. Here we just have a single column corresponding to A (so it's actually a vector rather than a matrix), and now we're just thinking about multiplying by a scalar x. And this defines a line through the origin; these are all the values that can be spanned by our single column of A.

Now, if I want to show that b *cannot* be spanned by this subspace, what I'm trying to show is that b is *not* on this line. What does that mean? It means it has a nonzero perpendicular part — if we decompose b into a part parallel and perpendicular to A, then that perpendicular part is nonzero. And that's what shows b is not on the line.



So if b is not in the subspace, that means it has a part that is perpendicular to the subspace. And it turns out the way that writes out algebraically is:

> **Claim 13.18 —** There is no solution to $Ax = b$ if and only if there exists y such that $yA = 0$ but $yb > 0$.

Proof. If there exists x such that $Ax = b$, then $yA = 0$ would imply that $yb = yAx = 0x = 0$. So if b is in the column space of A, then anything perpendicular to all of A is also perpendicular to b, because b is made up of pieces of A.

On the other hand, if there is no x such that $Ax = b$, that means the columns of A do not span b. This means there's a part of b that is perpendicular to the subspace $\{Ax\}$. And we can just let y be that perpendicular part. (So we basically subtract off everything in that subspace, and we're left with something perpendicular to it. It's still a portion of b, so its dot product with b is positive; but we've subtracted off everything in A, so it's now perpendicular to A.) □

§13.4.3 Duality

This is an interesting first example of *duality*. We started out with a linear system $Ax = b$ and asked whether it has a solution. And we said that when it *doesn't* have a solution, there's a *different* linear system which *does* have a solution.

> **Question 13.19.** Is y of polynomial size?

Well, remember that y satisfies $yA = 0$ and $yb > 0$. That means if we scale it, we have a solution to $yA = 0$ and $yb = 1$. (Whatever the value of the original yb was, we can scale y so that now the value is 1.) But this is just another linear system. And we've already argued that solutions to linear systems have polynomial size. So the answer is yes.

So there's a nice example of duality here — we started out with one linear system. If it has a solution, then great. But if it doesn't, then there's a dual that does have a solution, and vice versa. So *exactly one of these two linear systems has a solution*, and they prove each others' feasibility and infeasibility. Feasibility is easy to demonstrate, and since these are duals, you get the feasibility of one by showing the infeasibility of the other.

We've now resolved everything about linear systems — if you just have *equality* constraints, then this tells us the whole story. What makes linear programming interesting is that you throw in the inequality constraints $x \geq 0$. All of a sudden things get much harder. So we'll have to recapitulate many of these steps of understanding existence of a solution and whether it's polynomial size and how you verify it, but it'll take longer.

So we've shown that for any linear system, in polynomial time you can find a solution or prove there isn't one; and we can generate an easy-to-verify proof of non-solvability.

§13.5 Some geometric intuition

We've been doing a bunch of linear algebra, but now we'll do some geometry. This is only intuition, but it's helpful. The canonical form $Ax \leq b$ is an intersection of finitely many half-spaces; many people call such a thing a *polytope*. (Math communities have a lot of argument about what exactly the definitions are, but e.g., a D10 or D24 or D20 are all polytopes; you can imagine a box or some other shape.) In contrast, standard form starts with $Ax = b$, which defines a *subspace*. And then that subspace is bounded by the $x_i \geq 0$ hyperplanes. So from an intuitive perspective, you can think of canonical form as defining a sort of full-dimension thing (e.g., a box in 3 dimensions); meanwhile with standard form, you're creating some subspace and then chopping things off using the positive orthant. Both of these are actually fully general representations, so both intuitions are valid; but at different times it's useful to think of one or the other.

Either formulation is *convex* — if x and y are feasible for all the constraints, then so is $\lambda x + (1-\lambda)y$ for any $0 \leq \lambda \leq 1$. In other words, the line between x and y stays feasible (i.e., stays within the polytope).

§13.6 Corners

Now that we have a mental picture of what these polytopes look like, let's try to get a mental picture of what the optima look like.

Question 13.20. Given the polytope, where can the optimum be?

For some reason, we find it particularly easy to work with two-dimensional polytopes. Can a point in the middle be the optimum? Well, no — your objective function is linear, so you can imagine it as an arrow pointing in some direction, and then you can push the point in that direction. What about the point on the edge? You can still push it around along that edge, and one of the two directions won't decrease the objective. So the only possible optima are at 'corners.'

We need to characterize what these 'corners' are, and there's multiple ways to do so.

Definition 13.21. A vertex of a polytope is a point that can *not* be written as a convex combination of other feasible points.

Definition 13.22. An extreme point is a point that is a unique optimum for some linear objective.

For each of these corners, we can draw a particular objective function (pointing outwards from that corner) which makes that corner the only optimum in the polytope. That's not true for points on the edges. There is an objective function for which this point is optimal, but it's not the *unique* optimum.

Another way of thinking about vertices is that you can look at the inequalities that are actually equalities — you can look at which constraints are tight. That leads to the following definitions.

Definition 13.23. A constraint (either of the form $ax = b$ or $ax \geq b$) is tight (or active) at a point x if $ax = b$.

Definition 13.24. For an n-dimensional linear program (one over n scalar variables), a given point is said to be basic if all the equalities are tight, and in total, n linearly independent constraints are tight.

What does it mean for constraints to be linearly independent? Each constraint is defined by a vector, and you want those vectors to be independent.

In other words, this means x is the intersection of n linearly independent constraint planes.

Now, because x is at the intersection of n linearly independent constraints planes, that implies x is the *unique* point at which those constraint planes intersect (if we intersect n independent constraint planes, there's exactly one point that satisfies all of them).

Definition 13.25. A basic feasible solution (abbreviated BFS) is a solution that is basic and feasible.

If you look at the definition of basic solutions, you can have points that are outside the polytope (because they might violate some other constraints). So you have to add feasibility as well.

For the next few weeks, we should forget about the fact that BFS means 'breadth-first search'; it only means 'basic feasible solutions' for the time being.

Why did David give us three different notions of corners? Well, because they're all equivalent, and they're all useful (at different times we'll want to think about our corners in each of these ways).

Claim 13.26 — Any standard-form LP with an optimum has one at a basic feasible solution.

(Why did we add the qualifier 'with an optimum'? Well, some linear programs don't have optima — they may be infeasible or unbounded. But if there is one, then there's one at a BFS.)

Proof. Suppose we have a non-BFS optimum. We're going to transform it into an optimum at a BFS.

To do this, since it's a non-BFS, there are less than n (independent) tight constraints at x. That means there's at least one degree of freedom. More formally, the intersection of the constraints that are tight defines a subspace, and that subspace has dimension at least 1. In other words, we've got this optimum, and there are a few different hyperplanes going through it; their intersection forms a subspace. And there's fewer than n of them, so that subspace is not a single point; instead it's a subspace that contains a line.

Now, these are *all* the tight constraints at this point, and that means you can move on the line in either direction and stay feasible (at least for a little while).

So we've got this point, and we've got a line that we can move along. What happens to the objective function as we move along this line? The objective function is linear, so as you move along the line, the objective function has to change linearly; this means it decreases in one direction and increases in the other. But wait a minute! Can the objective function increase as we move along this line? No, because we're already at an optimum. This means it cannot increase in either direction. But that also means it can't decrease, because if it decreased in one direction then it would have to increase in the other direction.

So this implies that in fact, the whole line is optimal — we can move anywhere we want to along this line, and stay optimal.

So then we can move on the line in a direction that decreases some positive x_i. Recall that we're working with standard form, so all the x_i are constrained to be nonnegative; if they're all 0 then we're at the origin and the value of the objective is 0, which is uninteresting. But otherwise some x_i is positive; moving along this line means one direction will increase x_i, and the other will decrease.

So we move along this line in the direction of decreasing x_i. We can't do that indefinitely — we're eventually going to hit some constraint. We don't know which constraint we can hit, but the only constraints that are loose are the $x_j \geq 0$ constraints. So some x_j is going to reach 0.

So we started on a line and moved, and got to a new optimum with one more 0 in the vector. And we can repeat — we keep on zeroing out more and more values as long as we're not at a BFS. Eventually we'll either get to a BFS or the all-zeros point, which is also a BFS. So after iterating this at most n times, we will be at a BFS. □

This is actually one reason standard form is appealing — you can make the argument that you hit an $x_j \geq 0$ constraint. In contrast, this is not true for the canonical form — you can have a canonical form LP which does *not* have its optimum at a basic feasible solution.

Example 13.27

Consider the canonical-form LP over $\mathbb{R}^2$: maximize $0x + 1y$ subject to the constraint $y \leq 1$.

Here the objective says to go as high as we can, but we have a ceiling at $y = 1$, so the whole hyperplane $y = 1$ is optimum. So there is no BFS, but it's for kooky reasons — it's an unbounded polytope, so it goes on forever, and the trick we played of moving along a line might never stop. Standard form makes sure your polytope is at least halfway bounded — if you draw any line, it eventually has to leave the positive orthant, so it'll hit one of your constraints.

Corollary 13.28

If x is a feasible point, then we can find a BFS whose objective is at least as good as x.

If you go back and look at the proof, assuming you're starting at a non-optimal point, you can choose to move in the direction that improves the objective value, and you do this until you hit a constraint; you can then construct a BFS by adding tight constraints without making your objective worse.

Corollary 13.29

If the optimum is unique, then it is a BFS.

This is because there has to be an optimum at a BFS, so if there is only one then it's a BFS. In particular, this shows that every extreme point is a BFS (as it's the unique optimum for some objective function).

One question David will leave us with: he showed we can transform any linear program into a linear program in standard form, and that any linear program in standard form has one at a BFS. But we can start with a linear program that doesn't have any BFS, and somehow transforming it into standard form makes a BFS appear. And that's just true, and you can accept that if you want to; but thinking about it gives more insight into the transformations we've been doing.

Next time we'll talk more about how vertices and extreme points are the same thing, and then we'll really dive into constructing and verifying the optimum solution.

§14 October 11, 2024

§14.1 Review

We introduced linear programming last time, and talked about variables and constraints; we talked about feasible points, and optimality (finding a point with the best possible value of the objective function). We talked about how a linear program can be unbounded or bounded (in terms of the optimum) or infeasible.

We want to solve linear programs; we started by looking at linear systems. There, we showed that the solution to a linear system has size polynomial in the bits of the input numbers and the number of variables and constraints. On one hand, this is nice — there is a polynomial-sized solution. On the other hand, even if your input numbers are small, the output numbers could be very large (they could have polynomially many bits).

Then we saw a very interesting duality for linear systems.

Fact 14.1 — Either $Ax = b$ has a solution, or there exists y such that $yA = 0$ but $yb = 1$.

We originally stated this as $yb \neq 0$, but you can negate and scale to make it 1. We can also write this as

$$y \begin{bmatrix} A \\ b \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

So we have two linear systems such that exactly one has a solution; that means they're *dual* to each other.

Then we started talking about systems of linear *inequalities* (i.e., linear programs). We talked about how there's at least two elegant forms for linear programs: *standard form*, where we want to minimize cx subject to $Ax = b$ and $x \geq 0$. There's also *canonical form*, where we want to maximize yb subject to $yA \leq c$. The first is more elegant algorithmically, but the second is easier to think about. And every linear program can be converted into either.

§14.2 Characterizing the optima

Finally, we started talking about geometry and polytopes.

> **Question 14.2.** Where are optima?

We came up with three distinct characterizations of what you might call an optimum.

> **Definition 14.3.** A vertex is a feasible point which is not a convex combination of feasible points.

(If you have two feasible points, then any convex combination — i.e., a point on the line between them — is also feasible.)

> **Definition 14.4.** An extreme point is a point which is the unique optimum for *some* objective.

Note that we talk about extreme points of a polytope, which just has constraints (not an objective function); you figure out whether something is an extreme point by picking an objective function.

Finally, the weirdest of them:

> **Definition 14.5.** A basic feasible solution is a point which is feasible and *basic*, i.e., all linear equalities are tight at this point and n linearly independent constraints are tight (where n is the number of variables in the linear program).

Note that these n independent constraints include the linear equalities that are required to be tight — if you have lots of equality constraints, you only need a couple of other inequalities to be tight in order to get a basic solution.

In other words, this means your point is the unique intersection of n hyperplanes (one defined by each tight constraint).

In the end, we started arguing that all three of these definitions of corners are equivalent. We showed the following statement:

> **Proposition 14.6**
> Any standard-form linear program with an optimum has one at a basic feasible solution.

The LP may be infeasible and have no optima at all, but if there is one, there is one at a BFS.

Proof. The proof was by using a kind of *rounding scheme*. We start at any optimum, and if we don't have n tight constraints, then we move it to add more tight constraints until we have n. (There's a certain number

of constraints that are tight for your optimum; if there's fewer than n, you have a degree of freedom, and you can move in that direction. In standard form, you can move in the direction of $x_i = 0$; eventually that constraint or some other one becomes tight. And if we do this n times, we end up with n tight constraints, and get a BFS.) □

In fact, this proof showed the following statement.

Proposition 14.7

Given any point x, we can find a BFS whose objective value is no worse.

This is because we can always move in the direction of not making the objective worse, until we get to a BFS. This also shows the following:

Fact 14.8 — If the optimum is unique, then it is a BFS.

This is because we can always turn our optimum into a BFS, so if there's only one of them, it must already be a BFS.

Now we can use this to tie to the other two definitions. (These will be sketchy proofs, but should give us the idea.)

Proposition 14.9

A point is a BFS if and only if it is a vertex.

Proof. Suppose we're not a vertex. Then our point x is on a line between two feasible points a and b. Now, if our point is on a line between a and b that are both feasible, then can it be a basic feasible solution? No — if you can move on this line without violating feasibility in both directions, then you can't have n tight constraints (the intersection of all the tight constraints has to contain this line, so there must be fewer than n tight constraints). That implies your non-vertex is not a BFS.

Conversely, if you are not a BFS, which means you have less than n tight constraints, then can you be a vertex? No — there are fewer than n constraints that are tight at this point, which tells us that the intersection of all these tight constraints is at least a 1D subspace, i.e., it contains a line through our point x. But if it contains a line through x, then we can take one point on each side of x on the line that is feasible; and their convex combination will be equal to x. (We can sort of just move the same distance along the line in each direction, and then x is the midpoint, which is certainly a convex combination.)

So these two definitions are equivalent. □

Now let's look at extreme points — an extreme point is the unique optimum for some c. First suppose you're extreme; why does this mean you're a BFS? Well, we said earlier that if a linear program has an optimum, then it has one at a BFS. And here our extreme point is the *unique* optimum for a certain objective. So that means it must be a BFS — we know that *some* optimum is a BFS, and since this is the only optimum, we conclude that it is a BFS.

For the opposite direction, now we want to show that if we have a BFS, then it's an extreme point. Here we need to do a little bit of algebra; we'll focus on the case of standard form (but this argument generalizes). Suppose we have a BFS for a standard form LP; that means there are n tight constraints. Now, some of these are the equality constraints — standard form does come with equality constraints, and feasibility means you satisfy those. In a degenerate case, you might have n independent equality constraints; but that means there's only a unique feasible point, so it's clearly the unique optimum (for any linear program).

Otherwise, we'll have some inequalities that are tight; we represent this by a set T. Note that the only inequality constraints in a standard-form LP are those of the form $x_i \geq 0$. Then we define our objective as a sum of those tight constraints — i.e., as $\sum_{i \in T} x_i$.

We claim that the point we have is the unique point for which this objective is optimum. Note that the value of this objective is 0 at the BFS we started with. So let's consider any other x' that is feasible; we claim that it is not optimum. This is because it has to differ from x, which means it differs in some of the n independent constraints — if you had the same value on all of these constraints, then you'd be the same point. And if it's feasible, it can't differ on the equality constraints; so it has to differ on some of the *inequality* constraints. In other words, we must have $x'_j \neq 0$ for some $j \in T$.

And if $x'_j \neq 0$, what values can it take? Only positive values, because x' is feasible — so we actually have $x'_j > 0$. But this tells us the value of our objective function at x' is also greater than 0 — all the other x'_i are nonnegative (by the feasibility constraints) and we've just shown that one is positive, so their sum is positive. And that means x' is not optimum, because it's worse than our point that had a value of 0.

> **Remark 14.10.** We've proved this in standard form because it's nice to work with these simple constraints of the form $x_i \geq 0$, but you can do the same proof for a general LP; you're just looking at the distance from various constraints (here, that's just x_i).

So now we know all three are the same, and we can use the three terms interchangeably.

§14.3 Size of the optimum

We now have an understanding of where are the optima — or at least, where are *some* of the optima? They're at the corners of these polytopes. This actually tells us something very important — that we can *write down* an optimum of a linear program (assuming it exists). Why? The optimum is at an intersection of n *tight* linearly independent constraints — in other words, there's a row subset A' of A for which $A'x = b'$, and A' has n linearly independent rows. (Here we've gone back to the general representation of linear programs, where all the constraints are thrown into A; in standard form there are also some constraints $x_i \geq 0$, which would be represented here by 1's on the diagonal of A, and A' would be allowed to contain some of those as well as the equality constraints.) And so we know $x = (A')^{-1}b'$, which we proved last time has polynomial size.

So we can in fact write down the optimum solution for any linear program (that has one), which is progress.

> **Remark 14.11.** If you have n tight linearly independent constraints and I tell you the value of the point on each constraint, then you can recover the point using this inversion — in particular, two points can't have the same value on these n constraints.

§14.4 A first algorithm

So we can write down the answer. We've actually discovered something even more impressive — we can now give an algorithm for linear programming!

We know there's an optimum at the intersection of n independent tight constraints. So our first linear programming algorithm:

> **Algorithm 14.12**
>
> Try all possible subsets of n constraints. If they're linearly independent: find the unique point tight for all of them, check for feasibility, and compute the objective; we keep the best point over all subsets.

We need to check for feasibility because some subsets of constraints might intersect outside the polytope (and therefore we'd get a point, but it's not feasible).

How fast is this algorithm? If we have m constraints, then we need to examine $\binom{m}{n}$ possible sets of hyperplanes. And for each one, we need to do some checking (check for linear independence, and invert); all of this is basically n^3 work (matrix multiplication, matrix inversion, and Gaussian inversion all take roughly n^3 time — we'll have a $n \times n$ square matrix). There's also a size term, corresponding to the size of the numbers we work with.

This gives a *finite* time algorithm for linear programming! In fact, it's better than that — it's polynomial-time in *fixed dimension*. If I give you linear programs on the blackboard (i.e., in $\mathbb{R}^2$), you can solve them in polynomial time. This is actually important — in graphics you do a lot of 3-dimensional and 4-dimensional linear programs. In low dimensions linear programming actually becomes a geometry problem, and you can ask for efficient algorithms in low dimension; it turns out you can do way better than this (we'll see a bit of this when talking about computational geometry).

§14.5 Verifying optima

But of course the ultimate goal of being polynomial in both m *and* n is still far away. We're not going to jump directly towards that goal. Instead, we'll take another incremental step — we'll look at the decision problem of *verifying* optima.

> **Question 14.13.** If someone gives us an optimum, can we check that it is optimal?

That's got to be easier than actually finding the optimum, right? This is kind of like asking whether you can do linear programming in NP — if you're allowed to guess the solution and just have to check after you guess it, can you solve linear programming that way?

You can also ask the sort of converse question:

> **Question 14.14.** Given k, is the optimum value at least k?

That question is easily verifiable via a witness — if you want to prove to me a linear program has a solution whose value is better than some k, how do you do it (assuming you have unbounded computation)? You just take a point that is feasible and has value better than k (e.g., you can take the optimum); and now anyone who can do basic linear algebra can check that all the constraints are satisfied and the objective is greater than the target.

But what about the converse question?

> **Question 14.15.** Can I prove that there is *no* solution of value better than k? Can I give a certificate of *that*?

In slightly more standard terminology, given a minimization problem (which we'll put in standard form — to minimize cx subject to $Ax = b$ and $x \geq 0$), can I give a checkable lower bound on the optimum (any lower bound at all that is easy for someone to check)?

§14.6 Taking linear combinations

The way we'll approach this is similar to how we think about solving linear systems in preschool. You're given a system of linear equations you want to solve; what do you do? You try to take *combinations* of the equations to cancel out certain variables. In other words, you take linear combinations of some of your equations and that gives you new equations; and you try to make equations that are nicer by combining the equations you start with.

We're going to play the same trick here with our linear inequalities. To find the lower bound, we're going to play with the lower bounds we already have — we have these $Ax = b$ constraints and these $x \geq 0$ constraints. What can we do with them? Well, let's imagine we take some y, representing the coefficients for a linear combination of our equality constraints — in other words, we write $yA \cdot x = yb$. We started with some linear constraints, and we've made a new linear constraint that also has to be true. Any y that we choose will give us a valid linear equality constraint.

Now suppose that I get really lucky, and for the y that I chose, yA ends up equalling c. What would that mean? If $yA = c$, then what that shows is that $cx = yb$. But notice that this is true for *any* x. So if $cx = yb$ for every possible x, what does that mean about our linear program (which is trying to compute $\min cx$)? Remember that yb is just a number (we picked a particular y, and b is a vector from our linear program). So cx is always the same number; that means yb is the answer. (No matter which x you pick, the value cx of the objective is the value yb we computed.) So every feasible point has the same objective value — *all* the points are optima.

You'd have to be pretty lucky for that to happen — your linear program would be pretty weird (it would mean the objective is perpendicular to the $Ax = b$ subspace). But in general, that's not going to happen.

But suppose we settle for something less — suppose we find y such that $yA \leq c$ (coordinate-wise)? If $yA \leq c$, then since any feasible x satisfies $Ax = b$, we get

$$yb = y(Ax) = (yA)x.$$

And since x is feasible, all its coordinates are at least 0, which means this is at most cx (since yA is a vector component-wise less than c; we're multiplying each of them by a nonnegative number and adding them up; so this is at most what we'd get if every component was exactly c).

> **Remark 14.16.** We're going to be sloppy about writing transposes. (We're always taking dot products; some things, such as y, should be transposed. There are no outer products in this class; whenever you see two vectors being multiplied, assume the proper transposition.)

So if $yA \leq c$, then for any x, we'll have $yb \leq cx$. If we state that in English, it tells us that yb is a *lower bound* on cx for any feasible x. And that implies yb is a lower bound on OPT (the optimum value).

§14.7 The dual LP and weak duality

So we've given a procedure for constructing a lower bound on OPT — choose some y such that $yA \leq c$, and that gives a lower bound on OPT whose value is yb. Let's think like algorithmicists:

> **Question 14.17.** What should I do to choose the best possible lower bound?

To get the best lower bound, we want to optimize yb. So we should find the maximum possible value of yb that has the right sort of relationship to the constraints, namely that $yA \leq c$. What kind of problem is this? This is a LP!

So what we've just discovered is that one approach to finding a lower bound on your linear program is to solve a linear program. Why this might be useful is not yet clear, but it is an approach. For reasons that will later become clear, this is known as the *dual* linear program, and the original is known as the *primal* linear program.

What we've just proven:

> **Proposition 14.18** (Weak duality)
> The optimum of the dual (a maximization LP) is a lower bound on the optimum of the primal (a minimization LP).

So one LP is trying to push its objective down, the other is trying to push its objective up, and they block each other.

Note that this is symmetric — the primal LP provides an *upper* bound for the optimum of the dual LP, for the same reason. Another thing you can show (which we're not going to, because it involves a fair amount of algebra) is:

> **Fact 14.19 —** If you take the dual of the dual, you get back the primal.

(This is what makes it a *duality* relationship; the proof is in the lecture notes. It's a bit messy given what we know now because we've only talked about how to take the dual of a standard form LP, so you have to convert back and forth in the proof.)

This shows us some more things.

> **Claim 14.20 —** If the primal $\mathcal{P}$ is unbounded, then the dual $\mathcal{D}$ is infeasible.

This is because if the dual LP gave a solution, it would provide a bound on the primal. (Note that we're talking about unbounded *in objective value*, not about the polytope itself being unbounded.) Similarly, if $\mathcal{D}$ is unbounded, then $\mathcal{P}$ is infeasible.

Another thing that can happen is that possibly both are infeasible. And the last, and most normal case, is that $\mathcal{P}$ and $\mathcal{D}$ are both feasible and bounded.

> **Remark 14.21.** There's a few more combinations — there's 3 possibilities for the primal, and 3 for the dual, so there's 9 combinations. But actually only these four can occur. We've ruled out some of them with weak duality (if $\mathcal{P}$ is unbounded then $\mathcal{D}$ has to be infeasible), so 5 are left. But actually only these four can occur, and we'll see that shortly.

§14.8 Strong duality

Now we've derived weak duality — we've shown these two linear programs bound each other.

> **Question 14.22.** How tight are these bounds?

Now we get to the most general and powerful theorem in combinatorial optimization, which is *strong duality*.

> **Theorem 14.23** (Strong duality)
> If the primal $\mathcal{P}$ and dual $\mathcal{D}$ are both feasible, then their optima are equal.

So these bounds are as tight as you could possibly hope for. And both of these linear programs — the primal and dual — actually give you the same answer.

This completely resolves the problem of verifying possible solution values — if I want to prove to you that there is no solution of value better than k, then strong duality lets us do that — we find the optimum of the dual linear program, and this gives a tight bound on the objective value for the primal. We just check that value, and we know the primal cannot be better than that.

We're eventually going to give an algebraic proof of this, but now is the time when David admits he started college as a physics major. It didn't last, but he still every once in a while likes to get back to the physics perspective and use physical intuition in math. And there's a really beautiful physics proof of strong duality he'll show us, which will drive the algebraic proof we'll ultimately go through.

§14.8.1 A physics proof

We've got a standard form primal and canonical form dual. We'll start with the canonical form dual, and show that the primal has the same value as the dual we started with. We're also going to flip the sign of the objective (we'll see why soon) — so our dual is to minimize yb subject to $yA \geq c$.

What's the physics story of this going to be? The story is going to be that we take each of our constraints and we build a 'wall' or 'floor' in the direction of the constraint — each column of our linear program is the normal for one of these walls, and the position of the wall is determined by the corresponding element of c. We also orient b straight up.

So we've constructed this contraption; now what's a physical technique for finding the optimum? Well, we just drop the ball; and it's going to fall to the bottom of this well that we constructed from the constraints.

We are now going to do physics in order to understand why the ball stopped there.

So b is straight up; the columns a_i form constraint walls, and the ball now falls to the bottom. Let's call the point that it falls to y^*.

> **Question 14.24.** Why does the ball stop at y^* — why isn't it moving?

This has to be because there's 0 net force on the ball. So let's figure out the math of what that means. What forces are there on this ball? Well, there's gravity, corresponding to the (opposite) direction of b. There's also normal forces, which correspond to the a_i; and each of these forces has some magnitude x_i. And what does it mean that the net force is equal to 0? It means that $\sum a_i x_i = b$ (if we sum up all these wall-forces, they're equal to the force exerted by gravity) — or more compactly, $Ax = b$.

Can we say something more about these x_i — are they arbitrary? Well, they're nonnegative — we didn't put any magnets in these walls, so the walls are all pushing outwards (if we point all the constraints inwards).

And furthermore, $x_i = 0$ for each wall that's not touching the ball (since such a wall can't be exerting any force on the ball).

So what we've shown is that x is a feasible point for the polytope $Ax = b$ and $x \geq 0$. (You can see we're heading towards the primal we want — these force quantities are feasible for the primal polytope.) Now coming back to the issue of x_j being 0 when they're not touching, we can write this more mathematically by saying that $x_j = 0$ if $ya_j > c_j$ — this $ya_j > c_j$ condition is just a mathematical form of not touching. So if $ya_j - c_j$ is not 0, then $x_j = 0$. (This should remind us of complementary slackness.)

We can write this another way — this means

$$x_j(ya_j - c_j) = 0.$$

(We've just said one of these two quantities has to be 0, so their product is as well.)

But now we can move things around, and rewrite this as

$$x_j(ya_j) = x_j c_j.$$

So now that implies $cx = \sum_j x_j c_j = \sum_j x_j(ya_j) = (yA)x$. But that's equal to $y(Ax)$, and we just proved that $Ax = b$. So this is equal to yb.

So we've proven that the value of the primal objective is equal to the value of the dual objective; this proves strong duality. And if we now look backwards, what we relied on was the complementary slackness condition $x_j(ya_j - c_j) = 0$; this emerged from the physics that if you're not touching the ball, you can't be exerting a force on it.

Remark 14.25. Essentially what this says is when you look at the complementary solutions to the primal and dual, the only place where you have nonzero primal variables is where you have tight dual constraints. That's complementary slackness — if the dual constraint is not tight, then $x_j = 0$, so the primal constraint is tight. We'll go into much more detail about that next time.

Strong duality lets the two solutions prove each other's optimality — if I find an optimal solution to the primal and dual, I can look at them both and know they're both optimal, because they have the same objective value. So I can send you a LP and ask you to solve it and its dual; and I just need to check that your solutions are both feasible and have the same value. (The primal solution cannot be smaller than the dual, so if they're equal, then we must have the best possible value.)

Remark 14.26. Max-flow min-cut is a special case of LP duality, which we'll see next week. The feasible price function in min-cost flow is another example. The potentials used in shortest paths to get rid of negative weights are also an instance of LP duality.

Next week we'll give a formal proof of LP duality, which really just fills in the math of the physics analysis; we can construct it ourselves if we have nothing to do over the long weekend. Then we'll see the mechanics of taking a dual easily, and derive all these results like max-flow min-cut as special cases of duality. Then we'll start talking about *algorithms* for linear programming.

§15 October 16, 2024

Today we'll pick up on LP duality. Last time we saw a physics proof, today we'll see an actual proof; then we'll see how to take duals, and a bunch of applications.

§15.1 Recap of the physics proof

Last time we gave a physics *existence* proof — we showed there exists a dual solution with the same objective value as the primal. The rough sketch was that we started with a particular problem to minimize yb under the constraints $yA \geq c$. We showed that by looking at this as a problem involving a ball dropped down a well (where the constraints form the walls of the well, and b points straight upwards, so that the ball wants to drop as far as possible), we argued that the tight constraints correspond to nonzeor x_i 'forces' coming from the walls; while walls not touching the ball correspond to forces of 0. And all the x_i are at least 0 by physics. So we concluded that $Ax = b$ (again by physics — this is an equilibrium condition that the forces exerted on the ball should be balanced). So in other words, these forces x are feasible for the dual linear program, which is going to be a maximization problem — where we want to maximize cx subject to $Ax = b$ and $x \geq 0$. Just the fact that we're at the optimum of the primal meant we could get a feasible solution to the dual.

§15.1.1 Complementary slackness

In interesting thing we concluded here was *complementary slackness* — the assertion that $x_i = 0$ when the ball is not touching the wall. We can write this compactly as

$$(c - yA_i)x_i = 0.$$

(If we take the gap between the ball and the wall, and multiply it by the force, then we get 0 — if the ball is not touching the wall then the force is 0, and if it is then the first is 0.) If you have two solutions (one

for the primal and one for the dual) with complementary slackness, then as we'll see later, they're both optimum solutions. The way to see that is that summing the complementary slackness condition gives

$$cx = \sum c_i x_i = \sum yA_i x_i = yAx = yb.$$

This follows directly from complementary slackness — we don't need any of the rest of the physics. If we have complementary slackness, then the values of the two LPs are equal. And why does that mean x and y are optimal? The minimization LP provides an *upper bound* for the maximization one by *weak duality*; so if we have two where the value is equal, that means we're tight on both the upper and lower bound, so they must be optimal.

And this part is perfectly formal. So all we need to prove is that there exist complementary-slack solutions, and that will prove strong duality. That's what we're going to do more carefully.

§15.2 Formal proof of strong duality

Now let's dive into the formal proof. Again, we'll consider our optimum y for the primal problem to minimize yb subject to $yA \leq c$. (This proof is just formalizing the physics proof.)

We consider a maximal subset S of linearly independent tight constraints. When you have dependent constraints, it confuses the issue; if you have another constraint that is a combination of these constraints, you don't need to pay anything extra for it, because if these constraints are satisfied then that constraint is also automatically going to be satisfied. So we're basically throwing away constraints that don't matter — the other constraints are satisfied if these are (so they're only a distraction).

Notice that $|S| \leq m$, where m is the dimension of y (because these constraints are linearly independent).

Let A_S and c_S be the matrix and vector restricted to the appropriate columns (namely, those of S). Then A_S has full column rank (thanks to linear independence), and $yA_S = c_S$ (because we're just restricting to certain relevant columns).

> **Remark 15.1.** Note that A_S does not have to be a square matrix.

Also notice that yb is the optimum of the LP where we just restrict to these columns S — i.e., it's the minimum yb subject to $yA_S = c_S$. We had that all the constraints in S were tight. Is it possible that throwing away some of the other columns let us find a better y than this y that we started with that is tight at all these constraints? In theory, throwing away constraints could give us the possibility of making things better. But here it doesn't — the point is that the other constraints don't change anything. If you have $yA_S = c_S$, then thanks to the linear dependence, this means the other constraints are also satisfied. So any feasible solution to this LP is also a feasible solution to the other LP, which means it can't have a better value. So our y is still an optimal y for this cut-down LP. This is emphasizing that all we've done is remove some distractions — we haven't changed the linear program at all.

> **Remark 15.2.** In $yA = c$, each constraint is a column; so we're getting rid of some of the columns that are dependent on other columns, which corresponds to getting rid of constraints.

So it is sufficient to prove strong duality with respect to this smaller LP — if we find x that has the same objective value as this y, then we've proven strong duality for this LP, but also for the original. To add a bit more detail, if we do this, the dual linear program is $A_S x_S = b$ (each constraint in the primal problem gives a variable in the dual, and each variable in the primal gives a constraint in the dual — when you take a dual, you flip the role of variables and constraints) — we've thrown away some constraints in the primal problem, so some variables in the dual. Now suppose we've proven strong duality for this cut-down problem; then we have a set of variables x_S such that $yb = x_S c_S$ (that's what strong duality would show). Now if we have this dual solution to the cut-down LP, how do we generate a solution that satisfies strong duality

for the original LP? The other constraints didn't matter in the primal problem, so the other variables don't matter in the dual problem. So you can just set the other x_i's equal to 0 when $i \notin S$ — because we already have $yb = x_S c_S$, and if we just make the other x_i zero, this doesn't change the value of the right-hand side. (So this promotes our dual solution to the smaller LP to give strong duality for the original.)

So now we only need to prove strong duality in the case where the columns of A are linearly independent, and our solution y is tight everywhere — this again says that we only look at the walls that the ball is touching, and we've gotten rid of any redundant walls.

Now we're going to just assume that this is our original LP (and that we didn't have any of the original distracting linear constraints) — so the constraints are $yA = c$.

Now, just as in physics:

Claim 15.3 — There exists x such that $Ax = b$.

Proof. Suppose that this is not the case. Then we saw last week that this means there exists z such that $zA = 0$ but $zb \neq 0$ (that's duality for linear systems, and we're using that in the core of our proof of duality for linear inequalities). We can assume without loss of generality that $zb < 0$ (we can flip the sign of z if needed).

Now let's consider a new y' obtained as $y' = y + z$. What can we say about that solution?

First, we claim that it is feasible (for the primal LP $yA = c$). This is because $zA = 0$, so adding z doesn't affect yA (moving in the direction of z doesn't change your position with respect to any of the constraints) — i.e.,

$$y'A = (y + z)A = yA + zA = yA + 0.$$

But what can we say about $y'b$? Well, this shrinks — it's $yb + zb < yb$. And since we had a minimization problem, this means y' is better than y. We started with an optimal solution, and found a direction we could move in that improved this solution; this of course is a contradiction. □

This is just a formalization of the assertion that the forces have to be in balance — it says that if the forces are not in balance, then the ball can fall further (there's a direction it can move in that takes it further down).

Claim 15.4 — For this choice of x, we have $yb = cx$.

Proof. We know $Ax = b$, and $yA = c$. Combining these, we get

$$yb = y(Ax) = (yA)x = cx.$$ □

So the values of the two objectives are equal.

Now we just have to finish proving feasibility by showing that all the x_i's are nonnegative — that our forces are pushing on the ball rather than pulling on it.

Claim 15.5 — We have $x \geq 0$.

Proof. Assume for contradiction that $x_i < 0$. We again want to argue that then we can get a better value (we can improve the ball). If we think about it, if we've got a wall that's pulling on the ball, then the ball should just move in that direction — pushing on the ball stops it from moving, so pulling on the ball should help it move.

To formalize that mathematically, let $c' = c + e_i$, where e_i is the vector corresponding to the x_i which is negative. Let's consider a solution y' that satisfies $y'A = c'$. Going back to our physics, our walls were a physical representation of $yA \geq c$. What we've changed in the physical picture is that when we're solving $y'A = c'$ instead of $yA = c$, you're shifting the wall i up — we oriented all our forces so that moving the wall up tightens the constraint and makes c bigger. So solving $y'A = c'$ corresponds to moving the wall upwards. If I move one of these walls upwards, what happens to the objective function? Well, it's supposed to get worse, because I'm tightening the constraint.

And is there a solution to this linear system? Well, yes, because A has full column rank — the columns of A span everything, so I can get any c' I want by choosing an appropriate linear combination of the columns.

Note that y' is feasible for the original constraints — we took something that was tight on all the original constraints, and then we made one of the constraints tighter (we found something that's tight for the tighter constraint, so of course it also satisfies the original).

But now let's look at the value of the objective. We have $y'b = y'(Ax) = y'Ax = c'x$ because of how we constructed y'. Now what can we say about $c'x$? Well, we increased c_i and x_i was negative, so we're subtracting more, and we get something that is smaller than the original objective. □

Now we're done — we've shown by contradiction that we have x where $Ax = b$ and all the x_i are nonnegative, and we've shown that the value of the objective of this cut-down dual LP is the same as the value of the objective of the cut-down primal. So we've proven strong duality for the cut-down LP, and as we said earlier, if we now tack in a few 0's on the other x's, then we get strong duality for the original LP.

§15.3 Consequences

Now we're done with the math and we'll talk about the consequences. You start with this one LP that you may not know how to solve; and there's a dual you can construct, just by syntactic transformations, with the same answer. And maybe you can solve the dual; or maybe at least it gives you some insights to understand the primal better. We'll see now how useful this is.

So far we've talked about how to break down developing algorithms — you ask about the shape of the solution, then verifying feasibility, then finding a feasible solution, then verifying optimality. Strong duality gives us a very powerful way of verifying optimality. So the natural next thing would be to talk about algorithms for finding a feasible solution, before going on to talk about algorithms for finding optimal solutions. Unfortunately, strong duality shows that this is a useless question. It turns out that in linear programming, finding a feasible solution is exactly as hard as finding an optimal solution; and that comes from strong duality.

Corollary 15.6

Finding a feasible solution (in general) is as hard as finding an optimal one.

Proof. We'll show that if we have an algorithm for finding a feasible solution to any LP, then it can also find an optimal solution to any LP.

Given a LP you want to optimize (minimize yb subject to $yA \geq c$), we just find a feasible solution to the different LP $yA \geq c$, $Ax = b$, $x \geq 0$, $yb = cx$ (i.e., the original LP, the dual, and the constraint that the two are equal). Then any feasible solution consists of a point feasible for the primal and a point feasible for the dual where the objective values are equal, so you can conclude they're both optimal. □

This is weird. Finding a feasible flow or circulation or path is trivial, and optimizing them is hard. But in linear programming, just feasibility is as hard as optimization.

One other interesting insight that we skipped over: as we asw here, the x's in teh dual are kind of telling us how much the optimum of the primal will change if we move a constraint. Here we derived the contradictory case where if we have negative x_i, then the optimum changes in the wrong sign. But if we look at a non-contradictory case, we can inagne we have a ball and two walls defining where the ball is; moving one horizontal-ish constraint will cause a large change n the value of the objective, so this one will have a large x_i. Meanwhile, moving the other vertical-ish constraint will only cause a small change, so this x_i is going to be small. And in the limit, when you get a constraint that's not touching the ball or a constraint perpendicular to b, in that case the x_i is 0. So the dual variables kind of tell you about the sensitivity of the primal problem to the corresponding constraint.

§15.4 Rules for taking duals

Now we're going to take it as a given that duality is true, and we're going to use it.

The proof we did showed strong duality for canonical form linear programs — we started with a canonical form LP, and got a standard form LP where duality holds. We can transform any LP into canonical form and take its dual, but that's really painful; it's better to take any LP in any form and take its dual. So people figured out what happens to the original LP if you transform it and then take the dual.

If we consider a primal LP in general form, this means we want to minimize some objective value. In the standard form LP, we had a special role for nonnegativity constraints; those emerged from having constraints in the original LP and taking combinations. So unsurprisingly, these show up all the time. But thinking about the physics, if we had oriented the constraints in the opposite direction, then we would want the force to always be negative instead of positive. So we'll also have variables that need to be nonpositive. And finally, we'll have some variables that aren't constrained to be positive or negative.

So we'll assume three different kinds of variables; we'll suppose our objective function is $c_1x_1 + c_2x_2 + c_3x_3$ where $x_1 \geq 0$, $x_2 \leq 0$, and x_3 is UIS (unrestricted in sign). So we have three different kinds of variables.

We also have three different kinds of constraints — $=$, $\geq$, $\leq$. We write the first batch of constraints using three submatrices — as

$$A_{11}x_1 + A_{12}x_2 + A_{13}x_3 = b_1.$$

Then we do the same thing with other pieces of the matrix, in order to write down the $\geq$ and $\leq$ constraints —

$$A_{21}x_1 + A_{22}x_2 + A_{23}x_3 \geq b_2$$

and

$$A_{31}x_1 + A_{32}x_2 + A_{33}x_3 \leq b_3.$$

This captures all the possibilities.

Then the dual LP is going to be a maximization problem. We'll again have three types of variables y_1 (unrestricted), y_2 (nonnegative), and y_3 (positive); and the objective is $y_1b_1 + y_2b_2 + y_3b_3$.

And we transpose the whole matrix (or do left multiplication instead of right multiplication) — so we get

$$y_1A_{11} + y_2A_{21} + y_3A_{31} \leq c_1,$$

$$y_1A_{12} + y_2A_{22} + y_3A_{22} \geq c_2,$$

$$y_1A_{13} + y_2A_{23} + y_3A_{33} = c_3.$$

This is sort of a math salad (with lots of symbols and things), so let's write it in a nicer 'recipe' fashion that will help us also think about it more clearly.

Primal (min)		Dual (max)	
Constraints	$\geq b_i$	≥ 0	Variable
	$\leq b_i$	≤ 0	
	$= b_i$	UIS	
Variables	≥ 0	$\leq c_i$	Constraint
	≤ 0	$\geq c_i$	
	UIS	$= c_i$	

(An equality constraint is sort of saying the ball is forced to stay on the hyperplane; the force could go in either direction.)

You can remember this going back to the physics. A constraint $\geq b_i$ is pointed in the natural direction to prevent the thing from going down. If you have a constraint in the natural direction, then the corresponding force is nonnegative. If someone wrote the constraint backwards, so you have a minimization problem but the wall looks like it's trying to push things downwards, then the force has to be negative. And if you have an equality constraint, the ball has to live in the plane.

In the dual, the dual is a maximization problem, so the natural constraints are the $\leq c_i$ constraints; so those correspond to nonnegative variables. If you have constraints in the other direction, those again become nonpositive variables.

So you don't have to remember all the subscripts; you can just think about the relationship between each variable and its constraint.

Note that this table is only accurate because we're calling the minimization problem the primal and the maximization problem the dual; in general a $\geq$ in a minimization problem becomes a $\geq$ variable, and so on.

§15.5 Dual of shortest-paths

Now that we have the cookbook, we can start using it to take some duals; let's see what happens when we take the dual of our favorite problems. Note that this is completely mechanical, which means it's useful — if you're staring at a LP and don't know what to do, just take the dual. We just proved that in general this is pointless, but in fact you do often learn things by taking the dual — lots of advances in algorithms came from insights that arose from looking at the duals of the problems you were trying to solve.

It seems like a bit of an overkill to formulate shortest-paths as a LP (we can solve shortest-paths in near-linear time, and we still don't know how to solve LPs), but let's do it.

One way to physically interpret shortest paths is to make a graph with ropes, where the length of the rope corresponds to the weight of that edge; then you take the source and lift it up, and all the short paths will be tight based on these ropes.

If we write that down mathematically, if we're lifting up s, then some other vertex t is going to want to hang down as far as it can from s because of gravity. What's going to prevent it is the edge lengths. So the objective is to maximize $d_t - d_s$, subject to the constraints $d_j - d_i \leq c_{ij}$ (the triangle inequality).

We claim this is actually a formalization of the shortest path from s to t — separate s and t as far from each other as possible, subject to the triangle inequalities being satisfied on edges.

What's the dual of this going to look like? For this it's useful to make a picture of the constraint matrix, and ask, what are the rows and columns? Here there's a constraint per edge of the graph, and those correspond to the rows. Meanwhile the columns correspond to the nodes of the graph. (In LP language, the rows and columns correspond to constraints and variables; here we have a constraint for each edge, and a variable for each node.)

Now, what numbers are in this matrix? Only 1, 0, and -1 — we've got a constraint per edge that only hits two nodes, so in each row we'll have a $+1$ and a -1, and a whole lot of 0's elsewhere. The $+1$ is on the head of the edge, and the -1 is on the tail of the edge.

We've written this as a maximization problem, so in our table, this is going to be called the dual (it's the thing on the right-hand sie). Now we'll have what's called the primal. The first thing to ask is, what do the variables and constraints of the primal correspond to? Well, we had a variable per vertex in the starting problem (which we're calling the dual); that turns into a constraint per vertex in the primal. And similarly we have a variable per edge. Let's call these variables y_{ij} (or y_e).

Before getting to the constraints, what's the objective function? It's a minimization problem, and it's some linear function of the variables. What are the coefficients? Well, to answer this we have to figure out what the right-hand side of our constraints are; these are just the c_{ij} variables.

So then those c_{ij} are the coefficients of our linear combination — we want to minimize $\sum c_{ij}y_{ij}$.

Now we can think about the constraints. First, what kinds of variables are these, if we look at the table — are they nonnegative, nonpositive, or unrestricted in sign? We had a maximization problem, so $\leq$ is the natural constraint; natural constraints correspond to nonnegative forces, which means these variables are nonnegative.

Now we can talk about writing down the actual constraints. How do I take this matrix and read off one constraint? The dual problem was written in the form $Ad = \bullet$ (taking a constraint per row). In the primal, we put y on the other side, so now the constraints correspond to columns. So each column in the primal problem is going to become a constraint in the dual (columns corresponded to variables, and variables become constraints).

For a particular column associated with vertex j, what constraint do I read off in the primal problem — how do I figure out what numbers are going to appear in the column for vertex j? There's a 1 for every incoming edge, and a -1 for every outgoing edge — the rule for the rows said we put a 1 where the edge is incoming and a -1 where the edge is outgoing.

So that means our constraint corresponding to j says

$$\sum_{i\to j} y_{ij} - \sum_{j\to i} y_{ij}.$$

And what can we say about the bounds? These come from the coefficients of the objective function in the dual problem — objective coefficients become the value of the constraints. So over here, we should have a $+1$ if $j = t$, a -1 if $j = s$, and a 0 otherwise (because of all the 0 coefficients on all the other variables).

The last thing we have to ask is, do we want an equality, an inequality, or which way? The distances are unrestricted, so the constraints have to be equalities.

Okay, we've finished taking the dual. Well, this dual is another strange-looking linear program, but can we interpret it as something interesting? If we think of y_{ij} as the 'flow' on edge ij, then this LP is saying that the flow incoming minus the flow outgoing should be 0 unless you're s and t, $+1$ if you're t, and -1 if you're s. So this is saying to find a flow of value 1; and among those flows, what we're trying to do is minimize the total flow cost.

So the dual of shortest-paths is a min-cost flow problem. That's pretty nice, right?

Friday is a Jewish holiday, so we are going to repeat the 'show a video lecture and pause it to answer questions' strategy, in which we will take much more duals and maybe start an algorithm for solving LPs. But we have at least two more interesting duals to take — a dual of max-flow and of min-cost flow.

§16 October 18, 2024

§16.1 Review

Last class, we showed strong duality; we gave a physics proof two lectures ago, and last time we filled in the mathematics behind that physics proof. Then David took us through the cookbook of how we take

the dual of a linear program when you have a mixture of inequalities with upper and lower bounds and equalities, and variables that can be nonnegative, nonpositive, or unrestricted in sign. We wrote a whole table of mappings. We kind of decide to call the minimization problem the primal and the maximization the dual (this is an arbitrary decision).

Primal	min	max	Dual
Constraints	$\geq b_i$	≥ 0	Variables
	$\leq b_i$	≤ 0	
	$= b_i$	UIS	
Variables	≥ 0	$\leq c_i$	Constraints
	≤ 0	$\geq c_i$	
	UIS	$= c_i$	

If the constraint is in the natural direction — in the primal you have a minimization problem, so it's natural to imagine lower bounds — then you end up with the corresponding variable being nonnegative. This comes out of the physics proof — when the constraint is set up so that it's limiting the movement of the ball, the force is nonnegative from that constraint (it doesn't pull). But if you negated the constraint to turn it into an upper bound constraint, you'd negate the variable. And equality constraints say the ball has to be stuck to the hyperplane; then the force could be in either direction, depending on which way the ball is trying to move.

This is the duality cookbook table we saw last class. We also used strong duality to show that for linear programming, finding an optimum is completely equivalent to just finding a feasible point. This is shocking — in all the other problems we looked at, finding a feasible solution was easy. But strong duality says that just finding a feasible solution in general is as hard as optimizing.

But this is just *in general* — in specific cases there is a difference. There are many linear programs where it's trivial to find a feasible solution, but optimizing it is hard. How can these both be true? Well, the demonstration of equivalence requires taking your old linear program and constructing a *new* linear program. So even if it was easy to find a feasible point in the original linear program, the construction creates a new linear program where it's hard to find a feasible point.

But in practice, it's easy to find a feasible point; and this is a necessary starting point for a bunch of linear programs we'll be looking at soon. So they're theoretically equivalent, but different in practice.

Once we had the cookbook, we started using it. Last time we looked at the shortest paths problem, and took its dual; and we found that the dual of the shortest paths problem was the problem of finding 1 unit of flow at min-cost (from s to t). Once you know this, it makes a lot of sense — whatever the shortest path is, that's the cheapest way to send stuff from the source to the sink, so the cost of sending exactly one unit of stuff will be the length of the shortest path. So taking the dual reveals a different solution whose solution is the same.

This is common in linear programming; this is why when working on an optimization problem, it's often useful to formulate it as a linear program and take the dual; this can give you tremendous insight leading you to solve it a different way (even if you're not planning to do linear programming to solve the problem, it's useful for analytic insight).

§16.2 Dual of maximum flow

Today we'll see another example: we'll express max-flow as a linear program and take the dual, and then we'll stare at the dual and see what it tells us.

To formulate the max-flow problem, in order to make it more elegant, we'll turn it into a circulation problem: we'll add an edge $t \to s$, and we'll demand that we want to maximize the flow on the $t \to s$ edge. This is

equivalen to the max-flow problem — however much flow we can send from s to t, using this edge sends it back to s to get a circulation. The advantage of this is circulations are much more elegant because we no longer have exceptions to the balance constraints.

How do we formulate this as a linear program? The variables are goign to be the amount of flow on each edge. We'll write x_{vw} for the flow on the edge vw. Then our objective function is x_{ts}. And what are the constraints? There are the capacity constraints and flow conservation, so we get

$$\begin{aligned} \text{maximize} \quad & x_{ts} \\ \text{subject to} \quad & x_e \leq u_e \\ & \sum_w x_{vw} - x_{wv} = 0 \\ & x_e \geq 0. \end{aligned}$$

Now we need to think about how to take the dual of this linear program. Here we have variables and constraints, and we need to think about how to transform those in the dual. We called the max problem the dual, so here we're constructing the primal. And the transformation is that each variable in the dual gives a constraint in the primal, and each constraint in the dual gives a variable in the primal. We have two different types of constraints — the capacity constraints and the balance constraints. (In LP language, we don't think of $x_e \geq 0$ as a constraint — it's a sign restriction on the variables — so we don't turn this into a variable in the primal.) We'll use different variables for those kinds of constraints — we'll have variables z_v which correspond to the balance constraints (there is one balance constraint for each variable v, so one variable in the primal). We also need variables corresponding to the edge constraints. Again, here there's one constraint for each edge. So in the primal, there's going to be a variable y_e that corresponds to each capacity constraint.

Now, what kind of constraints are we going to have? You had variables for the flow on each edge, and each of those is going to get translated to a constraint. So there's a constraint per edge, which corresponds to the variable x_e.

Some of the dual LP is pretty easy to write down, because we know the coefficients of the objective and the right-hand sides of the constraints swap places. So what's the objective function going to be in the primal? At this point it's useful to start thinking about the matrix formulation of the constraints; the constraint matrix of this dual is multiplying the vector of variables x_e, and it has various constraints; and on the right-hand side is

$$\begin{bmatrix} | \\ u_e \\ | \\ | \\ 0 \\ | \end{bmatrix}$$

(we think of the top of the matrix as representing the capacity constraints, and the bottom as representing conservation constraints). And the upper part is $\leq$ constraints, and the lower part is $=$ constraints.

Finally, what about the quantities that go into this matrix? The top is going to be indexed by edges e, and the bottom by vertices v; and the columns will also be indexed by edges e.

Starting with the capacity constraints, what numbers get put into the matrix? The top is just an identity matrix, because each constraint deals with just one variable. And on the bottom, we have $+1$'s for incoming edges and -1's for outgoing edges, and a whole bunch of 0's elsewhere (for edges not incident to that vertex).

So we get

$$\left[\begin{array}{ccccc|c} 1 & & & & & \\ & 1 & & & & \\ & & 1 & & & \\ & & & 1 & & \\ \hline 0 & +1 & -1 & +1 & 0 & \\ +1 & -1 & 0 & 0 & 0 & \end{array}\right] \left[\begin{array}{c} | \\ | \\ x_e \\ | \\ | \end{array}\right]$$

Now that we have this picture, we can go back to thinking about the primal and what goes where. Where do we find the objective function of the primal? It's the bound on the dual. The primal problem should be a minimization problem. And the y_e's are going to go with the u_e's (the y_e's are the variables associated with the capacity constraints, and they get multiplied by the bounds on the capacity constraints), while the z_v's are getting multiplied by 0 (those are the associated bounds). So those don't show up, and the primal objective is just $\min \sum y_e u_e$.

Now, how do we write down the constraints/ We don't have to write down a lot of constraints, because there's just one kind — there's one constraint per edge. We need to figure out both the function and the bound; let's start with the function. For the constraint associated with edge e, we find the coefficients that determine that constraint in the coresponding column. (In the dual the constraints were rows; so when we go to the primal, we see constraints in the columns.) And the constraint associated with e is determined by the coefficients in column e.

And what are those coefficients? We'll see a bunch of 0's, so what are the nonzero coefficients? There's going to be a 1 for the associated capacity constraint (we're going to hit the diagonal of the identity matrix); that 1 comes from row e, so it corresponds to the variable matching that capacity constraint, namely y_e. So we get $1 \cdot y_e$ from the identity matrix part of the constraint .

What other nonzero things show up? We're going to see one $+1$ and one -1 — we put ± 1's where an edge is incident on a vertex. We have lots of ± 1's in a single row because a vertex has lots of incident edges. But in a column, each edge is only incident to two vertices — it goes into one and out of one — so there's a single $+1$ and a single -1. And these variables which should get ± 1's are the ones corresponding to the constraints; so if our edge is vw we're going to get a coefficient for z_v and z_w. So our constraint function is $1 \cdot y_{vw} + z_v - z_w$.

Next, what kind of constraint should this be — upper bound, lower bound, or equality? We had $x_e \geq 0$ (we're starting with nonnegative variables in the dual); so we're going to have a lower bound constraint here (according to our cookbook). (Nonnegative variables go to proper-direction constraints, and proper-direction constraints for a minimization problem are lower bounds.)

And finally, what goes on the right-hand side? For this, you look at the objective function of the original thing, which was $\max x_{ts}$; and that has a 1 on the ts edge and a 0 for everything else. So the only nonzero value on the right-hand side is going to be the one associated with the constraint on edge ts — we're going to get

$$y_{vw} + z_v - z_w \geq \begin{cases} 1 & \text{for } vw = ts \\ 0 & \text{for all other edges.} \end{cases}$$

Finally, what about the variables y and z — are they nonnegative, nonpositive, or unrestricted in sign? To figure out what the sign of y_e should be, we look at what type of inequality the constraint was. The corresponding constraint to y_e is the capacity constraint $x_e \leq u_e$; this is an upper bound, which is the proper direction for a maximization problem; and a variable corresponding to a proper-direction constraint is nonnegative. Similarly, for the z-variables, these correspond to the conservation constraints; those are equality constraints, so the z-variables are unrestricted in sign.

And now we're done; just by following the cookbook, we created a different linear program whose value is

exactly the same as the value of the original, namely

$$\begin{aligned} &\text{minimize} && \sum y_e u_e \\ &\text{subject to} && y_{vw} + z_v - z_w \geq \begin{cases} 1 & \text{for } vw = ts \\ 0 & \text{for all other edges.} \end{cases} \\ & && y_e \geq 0,\ z_v \text{ UIS.} \end{aligned}$$

§16.2.1 Interpreting the LP

Now we're going to try to reinterpret this as a different perspective on the original.

The creative insight starts by looking at the constraints $y_{vw} + z_v - z_w \geq 0$. That equation should remind us of the triangle inequality; and it suggests we should be thinking in terms of lengths and distances. So we can think of the y_{vw} as a length, and z_v as a distance; and then we have triangle inequalities as constraints.

We can observe a few more things about this. First, what can we say about y_{ts}? Well, when we added the edge u_{ts}, we needed to put infinite capacity on it; and if u_{ts} is infinite, then for any sane answer, y_{ts} is going to have to be 0 (or else the value is going to be infinite, and ∞ is certainly not the optimum of the original problem). So we need $y_{ts} = 0$, since $u_{ts} = \infty$.

And what does that mean? We had the special case constraint fo tthe ts edge; and that really says that

$$z_t - z_s \geq 1.$$

And if the y_e's are lengths and the z_v's are distances, then this is saying that the distance between the source and sink should be at least 1. In fact, you can also notice that the z's in this primal only show up subtracted from each other — changing them all by the same constant doesn't do anything. So in fact, we can subtract z_s from all of them and just fix $z_s = 0$. Then z_v is just the distance from s to v. So we have this constraint that says we want to force t and s to be separated by distances of at least 1.

Now, what do we get to choose? Our goal is to separate t and s, and we get to do so by choosing the lengths y_{vw} to create this separation. And what lengths do we want to choose? Well, we want to choose lengths that minimize this sum of length times capacity.

If you think about the pipes intuition for max-flow, capacity is kind of a cross-sectional area — how much can go through per unit time is determined by the cross-sectional area of the pipe. If you take that and multiply it by the length of the pipe, that gives you the volume of the pipe. So we want to pick lengths to minimize the total volume of the network. We don't want to put a lot of length on a fat pipe, because that creates a lot of volume; instead, we want to put our long lengths on thin pipes.

We're already starting to get an alternative perspective for max-flow. We were originally asking, what's the most flow you can send per unit time through the network? And what we're trying to do now is show that it's possible to separate s from t with a network that has relatively low volume. The volume of the network is what determines how much stuff can be in the network at any given time; if the volume is very small, then there's not much stuff in the network at any given time, so there can't be a lot of flow traversing the network.

Let's go a little deeper. One thing we'll point out is that this dual LP which says you want to separate s from t — that actually gives a justification for an algorithm we developed for max-flow. Blocking flows — or more generally, shortest augmenting paths) — was motivated by this. The whole idea was that they aimed to separate s and t by a large distance; and that's exactly what the dual linear program is trying to do. This is not uncommon — these are what you call *primal-dual algorithms*, algorithms driven by the structure of the dual problem. So when you're trying to solve an algorithmic problem, if you can formulate

it as a LP, often taking the dual gives you ideas about an algorithm for solving the problem (which is to sort of look at optimizing the dual).

Now let's ask, what would be a good solution to this problem — what's an example of how you might arrange for t and s to be separated by a distance of at least 1? What lengths might you assign?

One of the easiest things you can imagine doing is assigning only lengths of 0 and 1. If we focus our attention on doing this, where do we need to assign 1's in order to satisfy the constraints of the primal linear program? We need to put 1 unit of length on every path from s to t; and if we focus on the 0-1 case, this means we need to have an edge of length 1 on every path from s to t. And for every path from s to t to need to cross an edge of length 1, the edges of length 1 have to form a *cut*.

So one solution to this primal problem is to assign 1 to all edges on a cut, and 0 to all of the other edges — if we put 1 on every cut-edge, we've just made the distance from s to t 1. So any cut gives a feasible solution to the primal. And what's the best cut to use, given our objective function? Well, the value of our solution (for a given cut) is going to be the capacity of the cut. So if we're trying to optimize, the best thing to do is to use a min-cut — we put a 1 on every min-cut edge, and that gives you a feasible solution.

What have we just shown? We have weak duality, so we've just shown that

$$\text{max-flow} \leq \text{min-cut}.$$

Have we shown that they're equal? No — what we've shown is that there is *a* solution to the dual problem obtained by taking cuts, and of all these candidates, the best is the min-cut. But what if there's a fractional solution that gives an even better bound? So far we've only proven an upper bound — when you give a *specific* solution, that gives you a bound on the primal, but you don't get a guarantee on equality.

In this case, we will see in a moment that you can argue equality and recover max-flow min-cut; but we need some more machinery.

§16.2.2 Complementary slackness

So far, we've shown an upper-bound max-flow $\leq$ min-cut. But we restricted to 0-1 solutions; maybe there's a better *fractional* solution — a fractional cut that gives an even better bound on max-flow. We know from previous analysis that this isn't true; but today we're going to see how that comes out for free from linear programming, so that we don't have to be clever and prove it directly.

This requires us to introduce another idea, *complementary slackness*. We mentioned this in discussing min-cost flow; now we're going to see it in full generality.

Definition 16.1. Given primal and dual feasible solutions x (associated with the minimization problem) and y (associated with the maximization problem), we define their duality gap as $cx - yb$.

We'll always have $cx \geq yb$, which means the duality gap is always nonnegative. We also know that at the optima, the duality gap is 0. In fact, the optimum OPT is somewhere 'inside' the duality gap — it's between the upper and lower bounds we found from our arbitrary feasible solutions. So this duality gap in fact measures how far *both* solutions are from OPT — the two bounds sort of provide comparisons to each other (if they're very close to each other, you know both are very close to OPT).

Now if we assume our standard form-canonical form structure, we can rewrite the dual LP by turning it into standard form. The canonical form is $\max yb$ subject to $yA \leq c$. We're going to introduce *slack variables* $s \geq 0$, and write $yA + s = c$.

Now our duality gap $cx - yb$ can be written as

$$(yA + s)x - yb = yAx + sx - yb.$$

But in standard form, we have $Ax = b$; so this can be written as

$$yAx + sx - yb = yb + sx - yb = sx.$$

So our duality gap is a dot product between the primal variables and the dual slacks.

Another thing to notice is that in our standard form LP, all the x-variables are constrained to be nonnegative. We've also constrained $s \geq 0$. So if the duality gap is 0 (if we're at an optimum), that tells us that for each i, one of x_i and s_i has to be 0. (The only way to get 0 out of a sum of nonnegative terms is for every individual term to be 0 — so we need $s_i x_i = 0$ for every i, and the only way that can happen is if at least one of s_i and x_i is 0.)

This is known as the complementary slackness condition.

Theorem 16.2 (Complementary slackness)
Feasible solutions x and y are optimal if and only if for every i, one of x_i and $s_i = c_i - yA_i$ is 0.

In other words, either the primal variable is 0, or the dual constraint is tight. And that's the complementary slackness condition.

It's useful in both directions — you can *use* the complementary slackness condition to prove optimality (if this condition holds you're optimal). But more importantly, you can often use the fact that you're optimal to conclude that complementary slackness holds, and from that you can infer interesting things about your solution; and that's what we're going to do now.

Also note that this goes in both directions: at optimality, either a dual variable is 0, or the corresponding primal constraint is tight. And again, we phrased this in terms of standard and canonical form, but it's true for any form of linear program (you can generalize this).

§16.2.3 Proof of max-flow min-cut

Now let's apply this to max-flow. Consider some optimal solutions to the max-flow linear program and its dual (so we have variables x in the primal, and y and z in the dual). We're going to define a cut $\{v \mid z_v < 1\}$. First, we claim this is an s-t cut — this is because $z_s = 0 < 1$ so s will always be inside the set, while $z_t \geq 1$ so t will always be outside the set.

Now suppose that an edge vw leaves the cut — in other words, v is in the cut and w is not. That means $z_v < 1$ and $z_w = 1$. But now if we go back to the edge constraint, we had the triangle inequality constraint $y_{vw} + z_v - z_w \geq 0$. And from that, we can conclude that

$$y_{vw} \geq z_w - z_v > 0.$$

So we've just shown that a variable is not 0. What does complementary slackness tell us? Well, if we have a variable that's not 0, then the corresponding constraint has to be tight. And what is the corresponding constraint? It's the capacity constraint $x_{vw} \leq u_{vw}$. For this to be tight means that $x_{vw} = u_{vw}$ — in words, this says that the edge is saturated.

Let's look at the other direction (literally) — suppose vw *enters* the cut. What does that tell us? This means $z_v \geq 1$ and $z_w < 1$. And we also know that $y_{vw} \geq 0$ (that's one of the constraints). So that tells us

$$y_{vw} + z_v - z_w > 0.$$

In the language of complementary slackness, that's saying this dual constraint is not tight (we say it's *slack*). And that tells us the corresponding variable — which is x_{uv} — is 0. And in English, that says that that edge is empty.

In summary, in any optimal solution, there's a cut for which all edges exiting the cut are saturated and all entering edges are empty; and this tells us the value of the flow is equal to the value of this cut that we defined. So we have indeed recovered max-flow min-cut by looking at the optimum and defining a cut — complementary slackness tells us the value of the cut is equal to the value of the flow.

(Note that you might have an optimal fractional solution to the dual, but our *rounding procedure* turns it into a cut.)

§17 October 21, 2024

§17.1 Review

Last class, we got through most of the video lecture; we'll start this lecture by finishing up the bit that remained. We saw a duality cookbook table from a couple of lectures ago. Last time, we saw how we can take a max-flow problem and reformulate it as a circulation problem by adding an edge back from t to s (where we want to maximize the flow on that return edge). We saw how to take the dual, and we ended up with a kind of distance-setting linear program — where we needed to assign lengths y_e to edges e such that $d(s,t) \geq 1$, and we wanted to minimize the 'volume' $\sum y_e u_e$ of the resulting network (we think of the capacities u_e as the cross-sectional areas of the pipes, representing the rate at which things can flow; we multiply that by length to get a volume). So we want to minimize the volume of the network as long as we force s and t to be far apart. We have to make some edges long, but this gives us a penalty for making fat edges long; instead we want to make skinny edges long, as that contributes less volume.

We then introduced (or brought back) the notion of complementary slackness: when we have a primal and dual (in standard and canonical form), you can change the inequalities in the canonical form into equalities by adding *slack variables* representing how much slack you have in the constrants. And complementary slackness asserts that you can't have both a variable and the corresponding dual constraint slack at a certain point — at an optimum, one has to be tight. Conversely, if you satisfy complementary slackness, then you are at an optimum.

By using complementary slackness, we were able to deduce max-flow min-cut duality.

§17.2 Min-cost circulation

Next, we'll look at min-cost circulation. Fortunately, min-cost circulation is just a little bit different from max-flow. If we have a primal for max-flow as $\max x_{ts}$ subject to $\sum x_{vw} - x_{wv} = 0$, $x_{vw} \leq u_{vw}$, and $x \geq 0$, then how do we change it to become a min-cost circulation problem?

We've already gone to a circulation problem by adding back the reverse edge. When we add the costs, the only thing this changes is the objective function — instead of just putting a profit on the return edge, we're actually going to put costs onto the edges, which may motivate a different circulation.

So the only change to the primal LP is to chnage the objective from $\max x_{ts}$ to $\min \sum c_e x_e$.

This small change to the primal should result in a similarly small change to the dual. This changes just the right-hand side of the constraint. Remember that the coefficients of the objective in the primal become the bounds on the constraints in the dual. And in max-flow, those coefficients were all 0's except a single one, which meant the bounds were all 0's except a single one. Now we're changing those coefficients, so we change the bounds. There's one other small change: we're flipping from a maximization problem to a minimization problem. When we were doing a maximization problem, we had $\leq$ constraints, which were the natural direction. But now we've changed to a minimization problem and we still have $\leq$ constraints, so those are the unnatural direction. What this does is it changes the sign of the variables in the dual.

So we have to change the sign of the variables, and the bounds on the constraints. With that in mind, the entire dual is

$$\begin{aligned} \max \quad & \sum u_e y_e \\ \text{subject to } & z_v - z_w + y_{vw} \leq c_{vw} \\ & y \leq 0. \end{aligned}$$

(Again, thanks to the cookbook method, a small change to the primal immediately lets us derive the appropriate small change to the dual — when you add a variable to the primal, you're adding a constraint to the dual, and vice versa. Intuitively, this makes sense because adding a variable to the primal gives the primal more degrees of freedom, so it probably improves the primal. The dual action is to put a constraint on the dual, so you're restricting its degrees of freedom, and its objective is going to get worse. So this makes intuitive sense.)

How do we interpret this new dual that we computed? Well, it actually becomes clearer if we invert the signs of the z-variables; so we're going to define $p_v = -z_v$ and rewrite things. Now under this change of sign, we have

$$\begin{aligned} \max \quad & \sum u_e y_e \\ \text{subject to } & y_{vw} \leq c_{vw} + p_v - p_w \\ & y \leq 0. \end{aligned}$$

And the quantity $c_{vw} + p_v - p_w$ is something we've seen before — these are reduced costs under the prices p_v. So all this linear program is saying is that your y's need to be upper-bounded by the reduced costs, and also upper-bounded by 0. We can immediately see what value y should take on — if the reduced cost is negative, then y will be equal to the reduced cost. If the reduced cost is nonnegative, then the corresponding y will be zero. So

$$y_{vw} = \min\{\widetilde{c}_{vw}, 0\},$$

where $\widetilde{c}_{vw}$ denotes the reduced cost. (Every y wants to be as large as possible, since capacities are nonnegative; and these bounds are preventing it from doing so.)

What can we learn from this? Well, let's assume we have some optimums x^* and y^* for the primal and dual. What do we get from complementary slackness?

Suppose that $x^*_{vw} < u_{vw}$ (so our optimum flow is not saturating that edge). When you're looking at a slack constraint on the primal, that corresponds to a variable in the dual which must be zero. So this means $y_{vw} = 0$. And what can you infer from the fact that $y_{vw} = 0$, given that $y_{vw} = \min(\widetilde{c}_{vw}, 0)$? This implies $\widetilde{c}_{vw} \geq 0$ — because if it were negative, that would be the tight bound for y_{vw}, and y_{vw} could not be 0.

So a non-saturated edge must have nonnegative reduced cost. Turning this around, if $\widetilde{c}_{vw} < 0$, that implies we must have $x_{vw} = u_{vw}$ — if you have a negative reduced cost, then the edge is saturated.

Conversely, if $\widetilde{c}_{vw} > 0$, then by the definition of reduced costs, this means the constraint $y_{vw} \leq c_{vw} + p_v - p_w$ is slack — because y_{vw} is not allowed to be positive, so if $\widetilde{c}_{vw}$ is positive then $y \neq \widetilde{c}_{vw}$, which means this constraint is slack. But that implies the corresponding primal variable is 0. And the corresponding primal variable is x_{vw}; so this means $x_{vw} = 0$. In ot

her words, if $\widetilde{c}_{vw} > 0$, then the flow is 0.

So the complementary slackness that we derived laboriously about min-cost flow is just a special case of that in linear programming — if the reduced cost is negative then the edge is saturated, and if it's positive then the edge has no flow. So everything comes with linear programming — it's this great big sledgehammer.

Student Question. *Is this kind of analysis typically useful for algorithms?*

Answer. Yes, for several reasons. It often gives you insight about the problem — David wouldn't be surprised if Ford–Fulkerson developed Max-Flow Min-Cut duality by looking at the dual (which involves distances, and a cut is a good way of generating distances). It also has lots of algorithmic ramifications. There's an entire class of algorithmic techniques known as *primal-dual* techniques. Their point is that as we've seen here through complementary slackness, the solutions to the primal and dual problem inform each other. Last time we saw duality, we saw the dual variables kind of talk about how sensitive the primal linear progrma is to changes in the value of a constraint; so there are these connections. And in many cases, it turns out the best way to find a primal solution is to simultaneously work towards a solution to the primal and dual, and use the not-yet-optimal solution to one of them in order to guide you as to how to improve the solution you have to the other. This is again complementary slackness — if in your dual one variable is very far from 0, that's a hint that the primal wants that constraint to be tight. So these things steer each other.

§17.3 Algorithms for linear programming

What's left? Well, we still haven't talked about algorithms for linear programming; now it's time to do so.

At a high level, there are three classes of algorithms for linear programming. The first is the *simplex* algorithm, developed in the 1940s by George Dantzig (as part of the WW2 effort to optimize the army and provisions and transportation and so on). It's hard to think of an algorithm that has had more impact over now 70 years. It works great in practice — except when it doesn't. The problem is that we can't really know in advance whether it is going to work in practice or not, because it's not polynomial — it has worst-case exponential runtimes. Nevertheless, it's still powerful in practice, and Dantzig is sort of considered the father of mathematical optimization.

But this left open the question:

Question 17.1. Is there a polynomial-time algorithm for linear programming?

The answer was proven to be yes through the development of the *ellipsoid* algorithm (from the 1970s). This is amazing in terms of theory. It's polynomial time, but it's n^6, which is not a good idea for practice. But the theory ramifications of this are huge, and we will mention a few of them when we talk about the method.

Question 17.2. We've got a theoretically polynomial time algorithm, but what about in practice?

That was answered in the affirmative in the 1980s through *interior point* algorithms. Initially these were just another interesting theoretical polynomial time algorithm. But over the course of decades, people were able to take insights and develop actual practical interior point algorithms, which have both worst-case polynomial bounds and are great in practice, and often outperform simplex (even on practical problems). There's been a tremendous amount of research; this all falls into the space of continuous optimization. The earlier work was on linear programming in general, but in a sense all the recent breakthroughs on max-flow and such are basically interior point algorithms.

We're going to formally present the simplex algorithm, give the main ideas of ellipsoid relatively mathematically (so we can believe it's true) but not fill in the details (there's an amazing book called Combinatorial Optimization that takes a whole book to fill in the details); and for interior point we'll just draw a few pretty pictures.

§17.4 The simplex algorithm

Simplex is a local search algorithm. It starts on some feasible point — specifically, a vertex, because we know there's always an optimum at a vertex — and looks for a better vertex. It repeats this until it can't find one, and then it says I'm at the optimum.

All of the details are in how one finds the improving vertex. Pictorially, going back to physics, how do you get the ball to the bottom of the well? Well, you just hold it over the polytope and drop it. As it falls, it's going to hit a constraint and roll down it and hit another constraint and get caught in the gutter formed by those two, and then roll in that direction until it hits another constraint, and then it'll change direction and keep rolling down; and so on.

We basically just need to do the math of the ball rolling down the gutters, and the place where it hits a constraint and changes direction to roll down a different edge.

§17.4.1 Characterizing vertices

For this, we're going to restructure our standard form LP

$$\begin{aligned} \min cx \\ Ax = b \\ x \geq 0. \end{aligned}$$

Without loss of generality, we'll claim that A has full row rank — in other words, there are no linearly dependent subsets of the rows. Why? If we start with a basis of rows $a_1, \ldots, a_m$, then if we have any other constraint that is a linear combination of these (meaning that $a_i = \sum \lambda_i a_i$ using this basis), then of course $a_\ell x = \sum \lambda_i a_i x$. But this is $\sum \lambda_i b_i$, which means that we had better have $\sum \lambda_i b_i = b_\ell$, or else there won't be any feasible solution at all. So then once we're feasible for all the independent rows, we're also feasible for any dependent row. And so we can just ignore all those dependent rows, and pretend they were never there to begin with.

Now let's look at a particular vertex. Given that we have full row rank, we can sort of get a better picture of what vertices are. What do we know about vertices? They have to have n tight linearly independent constraints. We know our m equality constraints are all tight (by definition, if our point is feasible). So in order to make a vertex, we need $n - m$ additional constraints (that are linearly independent) to be tight. And we find those from the nonnegativity constraints — so $n - m$ of the $x_i \geq 0$ constraints have to be tight, meaning those x_i have to be 0. (Some extra ones might be 0 as well; that doesn't matter.) And together, all these constraints have to form a basis.

Now, we got a good start on that by making sure our rows are linearly independent, so that we can take all the rows as the start of the basis; then it's just a question of putting in some other x_i's that finish the basis.

Let's draw a picture of that — let's write the matrix of tight constraints. We've got the m rows of A, and then to add in the tight constraints, well, those just select out particular x_i's. How many do we have? Well, we have $n - m$ of the $x_i \geq 0$ constraints that are tight; and A stretches all the way out. So we'll have the picture

$$\begin{bmatrix} - & A & - \\ I & & 0 \end{bmatrix} x = \begin{bmatrix} b \\ 0 \end{bmatrix}.$$

What does this mean for all these rows to form a basis? We need the rows to be linearly independent; but this is equivalent to the *columns* being linearly independent (note that this is a $n \times n$ matrix; so it's nonsingular if it has n linearly independent columns).

And what conditions need to be satisfied in order for the columns to be linearly independent?

First, the m columns of A on the right (which go with the 0's) have to be linearly independent of each other — if any subset of these m columns was dependent, throwing in 0's doesn't change that. So these columns on the right have to be linearly independent.

And what are those columns? Each column corresponds to one of the x-variables. So the columns over the 0's are the columns that correspond to the nonzero x_i — because the columns on the left are the ones that constrain some x_i to be 0, so the columns on the right are the ones where we're allowing the x_i to be positive.

So the parts of A corresponding to those x_i have to be linearly independent. What else has to be true? Well, nothing really — the $n - m$ columns on the left are obviously independent from each other and the remainder (since they have 1's on the bottom rows, and there's no way to cancel these out). So this part of the matrix is linearly independent no matter what you add to it; and the *only* thing that's necessary is that you have linear independence of the m columns on the right.

This gives another way to define a vertex:

Fact 17.3 — A point x is a vertex if and only if $Ax = b$, and m linearly independent columns of A include all those corresponding to variables x_j that are not 0.

Definition 17.4. This set of m columns is called basic. The set of corresponding x_j's are called basic; and the others are called nonbasic.

It's not a basis of the entire space, of course (but it is a basis of $\mathbb{R}^m$).

We'll use B and N to denote the basic and nonbasic variables.

Simplex is all about playing with this basis. Note that if I give you the basis, you can compute the vertex x that the basis came from. Why is that? Well, remember that A_B (where we restirct A to the rows and columns B of A), that's a $m \times m$ full-rank matrix (there are m rows, and we're picking out m independent columns). We also have that if we separate out the non-basic and basic columns, then

$$\begin{bmatrix} A_N & A_B \end{bmatrix} \begin{bmatrix} x_N \\ x_B \end{bmatrix} = b.$$

But x_N are the non-basic variables, and that means they're all 0's. So what this actually tells us is that A_N and x_N play no role, and really all that's going on is that

$$A_B x_B = b.$$

And we just said that A_B has full rank, so that tells us that $x_B = A_B^{-1} b$. And of course $x_N = 0$. So just from knowing what the basic columns are, we can reconstruct what x is.

The summary here is:

Lemma 17.5

A point x is a vertex (i.e., a BFS) if there exists a basis B such that $x_N = 0$, A_B is nonsingular, and $x_B = A_B^{-1} b$ is nonnegative.

(We know what the x_B is, so we just have to require it to be feasible.) So all of this work was to give a different definition of a vertex that is particularly useful for the simplex algorithm.

§17.4.2 The simplex algorithm

The algorithm is to start at a basis (equivalently, a vertex; we'll use these interchangeably) and try to move to a better one. We'll argue that if you can't, then you're actually optimal.

Let's fill in some math of what it means to try to move to a better basis. Everything will be discussed relative to our current x and corresponding basis B.

First, we're going to rewrite the LP — or expand it — as $\min c_B x_B + c_N x_N$ subject to the constraints $A_B x_B + A_N x_N = b$ and $x \geq 0$ (this is just a rewrite of the standard form LP). Note that this is really just a subscripting trick — we're just taking any feasible x and looking separately at the basic and nonbasic variables.

This constraint means $A_B x_B = b - A_N x_N$. And since we know A_B is nonsingular, we can multiply that out and say that

$$x_B = A_B^{-1}(b - A_N x_N).$$

Notice that this is true for *any* feasible x; we're not only saying this about our current vertex. It's just another way of expressing feasibility. (The non-basic variables are all 0 *at our particular vertex*, but this formula is true for any feasible x, including ones that don't have 0's in x_N.)

We can take that and plug it into the objective function; we can write $cx = c_B x_B + c_N x_N$, and we just expressed x_B as a function of x_N, so we can plug that in; this tells us

$$cx = c_B A_B^{-1}(b - A_N x_N) + c_N x_N.$$

We can split this into a constant term and some stuff involving x_N, as

$$cx = c_B A_B^{-1} b + (-c_B A_B^{-1} A_N + c_N) x_N.$$

Again, this is a formula that is true for *any* feasible x.

The first term is just a constant, so we can ignore it. Meanwhile, we can take the big, messy formula on the right, and write it as $\widetilde{c}_N x_N$, where $\widetilde{c}_N = -c_B A_B^{-1} A_N + c_N$. We'll call these *reduced costs*. (And yes, they work just like reduced costs in min-cost flow and shortest paths — they're a transformation that has not changed the problem, but has made some things apparent that were not previously apparent, which are useful for both algorithms and analysis.)

How do we use this? Essentially this is saying that if you give me a feasible x, I have a way to calculate the objective that involves only looking at the non-basic variables (you don't even have to tell me what the basic variables are). Now suppose that *all* of these reduced costs are nonnegative (i.e., $\widetilde{c}_j \geq 0$ for all j). What is the implication? Well, let's remember that at our current vertex, all the non-basic variables are 0. So the value of the objective at our current vertex is just the constant $c_B A_B^{-1} b$ — all the rest gets cancelled out by the 0's of the current vertex. What if all the reduced costs were nonnegative — what does that tell us about any other feasible point? It means that any other feasible point — if that feasible point is 0 in all the non-basic variables, then it's our vertex, not a different point. So the only way to be a different point is to be nonzero in any of the non-basic variables. But that isn't going to contribute negatively to this $\widetilde{c}_N x_N$ term, which means your value is no better than our current vertex. So any other vertex has objective at least that constant term $c_B A_B^{-1} b$, and is not better than our current vertex. So we're at OPT.

Again, reduced costs make it very straightforward to notice that you are at OPT, just as in min-cost flow.

On the other hand, suppose that some reduced cost is negative — meaning $\widetilde{c}_j < 0$. What opportunity does that give us — how might we improve our objective if we have a negative reduced cost? Increasing x_j will improve the objective. *Can* we increase x_j? What we just argued is that the basic variables are a function of the non-basic variables. The non-basic variables don't interfere with each other at all — we can set those however we want, and then compute a value for the basic variables. Doing this increase is going to change x_B, because it's a function of the non-basic variables. We can't necessarily keep changing x_j indefinitely, because the x_B's are going to change in return. But we can increase x_j until *some* x_i in the basis becomes tight (meaning it becomes 0). And that's a point at which we run into a new constraint that prevents us from moving further.

To picture this, imagine we have some kidn of cube-shaped constraints; we start with x at some corner. Modifying x_j is going to cause a change in a bunch of other basic variables; we can continue until some other constraint becomes tight, meaning some x_i in the basis becomes 0. And figuring out which one of the basic variables gets tight first is some straightforward linear algebra.

This is called a *pivot* step. Notice that we increased some non-basic variable from 0 — x_j (from our non-basic set) becomes slack. But if it becomes slack, it has to leave the non-basic set and move to the basic set. Meanwhile, some x_i that was in the basic set can now move to the non-basic set. So that's the pivot step — you're bringing one variable out of the basis, and putting a replacement variable into the basis.

Remark 17.6. After a pivot step (just as in min-cost flow), you'll have to recompute new reduced costs, based on the new vertex that you are now at.

Student Question. *Do we know the new columns will be independent?*

Answer. Yes, though we're not going to prove it. The idea is that because this constraint is becoming tight, it can't be dependent — it has to be in some other direction.

§17.4.3 Potential issues

This is the core idea of simplex, but there are some issues.

First, in order to run simplex, you need an initial vertex — because you have to pivot *from* somewhere. How do you find that? Well, you solve it by solving a linear program. But that is at least an easier linear program to solve.

This is a bit weird because finding an initial vertex is really the LP feasibility problem, which we said is no easier than LP optimization. But on the homework we'll see this is not actually a paradox, and you can make it work.

The next problem is the really serious problem. We made things sound great — you take one variable out of the basis and move, so you've improved on your basis. Clearly this will take us to the optimum, right? If we keep improving, then we can't improve forever; and there's only a finite number of vertices, so after a finite number of steps, we'll be at the optimum vertex, right?

But there's something we've said that's not quite true. Must a pivot step actually improve your objective? What can go wrong is when you move until some other $x_j = 0$ constraint becomes tight, maybe it's already tight — maybe you had many constraints that were already tight at that vertex but were not in the basis. There's nothing to stop your linear program from having lots of constraints that all meet at the same vertex.

So the problem is that maybe the improvement distance is equal to 0, because that other $x_i \geq 0$ constraint is already tight.

Then you can still do the pivot step — we had an old constraint in the basis which was tight, and we do a pivot, which takes that constraint out of the basis and puts some other constraint into the basis. But since the new constraint was already tight, we don't actually move; we just have a different basis that corresponds to exactly the same vertex. And if you do that once, the next time you might do a next pivot that brings the old basis back.

So the point is that you're sometimes forced to make pivots that don't improve. And if you do that, you may get cycling pivots, where you keep on changing your representation of the vertex without actually moving to a different vertex. This again happens because you have too many constraints intersecting at the same spot.

We won't go into the details, but the point is that if you take the right subset of constraints, then you're in great shape (this will reveal when you can move to an edge). But you don't know which subset is that proper subset that reveals things.

The way out of this is that you *can* prove that non-cycling pivot rules exist. There are rules for deciding on your pivot that guarantee you will never go back to a basis you've used before. The most commonly discussed one is to use lexicographic ordering — when you have multiple choices about which variables to bring into your basis, you aim for the lexicographically first basis. And you can show that you will never have to go back to a lexicographically worse basis.

From a theoretical perspective, it's easier to think about perturbation. Imagine you take your LP and add an infinitestimal perturbation to each constraint. Then it will no longer be the case that all these constraints intersect at a single point — there will be only n — and therefore there will always be movement, every time you switch bases. In fact, lexicographic order is really just an implementation of perturbation, where you use different orders of ε for different variables.

So that's great; that means you can make simplex *terminate*.

§17.4.4 Runtime of simplex

Question 17.7. How long is it going to take for simplex to terminate?

Well, that depends on how many vertices you visit. In the worst case, you could imagine that maybe you visit all vertices. And how many vertices might there be in a m-constraint n-dimensional polytope? There's $\binom{n}{m}$. And maybe you'll visit too many, which makes the algorithm slow.

Unfortunately we don't know how to get around this problem, and simplex is slow in theory. In fact, Klee and Minty developed the Klee–Minty cube, which is a standard hypercube in n dimensions twisted a bit; and this twist is such that a simplex algorithm visits *every single vertex*, if you simply choose pivots greedily. Simplex is a meta-algorithm, and Klee–Minty only proved this result for a particular pivot rule; but no pivot rule that has been formally analyzed avoids this problem.

One thing you might ask is, maybe at least the *diameter* of the polytope is small? Simplex is traversing between a bunch of vertices along edges; maybe there's some powerful result guaranteeing the polytope has small diameter. This doesn't necessarily guarantee a fast simplex algorithm, because these short paths might not actually go along the simplex algorithm (maybe to get from vertex A to B, you'd have to temporarily make the objective worse). But if the diameter is large, there's no hope for simplex to be fast; if the diameter is small, then at least there's hope for it to work with a good pivot rule.

Conjecture 17.8 (Hirsch 1957) **—** The dimaneter o a polytope is at most $m + n$.

(It's more that it's harmful for the diameter to be large — the simplex algorithm is traversing a path on the polytope, so if the diameter is large, then simplex cannot be fast.)

This conjecture was attacked for decades without any success, until in 2010, it was *disproved* by Santos, who proved that the diameter can be at least $(1+\varepsilon)(m+n)$. This is a really cool result but does not change anything; so it is still conceivable that there's a fast simplex algorithm.

On the positive side, Kalai–Kleitman (1992) gave a bound of $m^{\log n}$, which is a tremendous improvement vs. the m^n that we get just by counting vertices. The way they proved this was by using a cool technique of randomization, which we can talk about another time (e.g., next fall). This randomization idea has actually led to some breakthrough algorithms. At this point, there is alinear time linear programming algorithm in fixed dimension. So if n is a constant, then you can solve linear programming in linear time. And the algorithm is incredibly simple, although brilliant. (Seidel 1990s developed the first one, and David will teach it next fall.)

But we're so close — if instead of $\log n$ we had a constant, we'd have polynomial time linear programming. But it's not a constant, so we don't. (The fixed-dimension case is a small number of variables and large number of constraints, which happens all the time so is important for practice; but the big theory question is still open.)

Despite the weakness of the theory, simplex has had tremendous value in practice, so it's a really good thing to know.

Next time we'll talk about ellipsoid and interior point, and then we'll start approximation algorithms.

§18 October 23, 2024

Facilities claims that they have warmed up the room.

§18.1 The simplex algorithm and duality

Last class, when we developed the simplex algorithm, we defined the *reduced costs* on the nonbasic variables by the formula

$$\widetilde{c}_N = c_N - c_B A_B^{-1} A_N.$$

This assigned a new objective coefficient to every nonbasic variable. And we argued that if $\widetilde{c}_N \geq 0$, then we're at an optimum — because any change to the nonbasic variables is going to increase the value of the objective, which makes it worse. So that tells us the place we're at cannot be improved on.

On the other hand, if $\widetilde{c}_j < 0$, we said you can *pivot*. Intuitively, this takes you to a vertex with an improved objective value, and that's how simplex works. But there are all these weird cases where you have a pivot that's not able to move, but just changes the basis. Then you have questions about whether this terminates or whether we cycle through pivots. There are non-cycling pivot rules that guarantee you never repeat a basis, so you'll eventually end up at an optimum, but it could take exponential time.

There's one other thing to say about these reduced costs. Suppose we are at an optimum, where $\widetilde{c}_N \geq 0$. We can then define a particular point y by the formula $y = c_B A_B^{-1}$. (This is right here in the definition of the reduced costs.) Since we defined it that way, that implies

$$\widetilde{c}_N = c_B - yA_N.$$

But we just said that we're at OPT, so this is nonnegative. We can then move things over and conclude that $yA_N \leq c_N$.

Now, from the definition of y, we also have that $yA_B = c_B$. So if we put these together, what we conclude is that $yA \leq c$. So what does that tell us about y? It's feasible in the dual.

Also, what's yb? Well, we have a formula for y, and we can plug that in; we get

$$yb = c_B A_B^{-1} b.$$

But $A_B^{-1}b$ is just x_B. (We said that the basis defines x, and this is the formula by which that works.) So we get $yb = c_B x_B$. And of course $cx = c_B x_B + c_N x_N$, and this second term is 0 (because $x_N = 0$). So what we've found is that $yb = cx$. And what *that* tells us is that it's optimal — so we have a dual feasible solution y whose objective value is equal to the primal objective value. And this proves that both the primal and the dual are optimal.

So in fact, the simplex algorithm is solving the primal problem and the dual problem at the same time — at the moment you get your primal optimal solution, you also get your dual optimal solution.

And you can always define y (for any x, not just an optimum one) — so while you're running your simplex algorithm and updating x, you can also look at this y that's updating at the same time. So more generally, with your primal x, you have this dual y. And the *difference* $yb - cx$ is the *duality gap*, which is a very useful measure of your progress — how close you're getting to the optimum (because we know the optimum

value is somewhere in between — so as this gap shrinks, you're honing in closer and closer to the value of your linear program).

In fact, this gives a different proof that when the simplex algorithm stops, it is at OPT. We earlier saw a proof based on reduced costs and arguing that moving anywhere would make the objective worse; but this is a separate argument that just says that if you've stopped, then we can define this y which is dual feasible and whose values are the same; this proves simplex only stops at an optimum value.

This, interestingly enough, gives another way to prove strong duality — which is to show that some pivoting rule is non-cycling. If there is a non-cycling pivot rule, then the simplex algorithm will eventually run out of vertices, and then it will stop. And then we can conclude that it's optimum based on the equality of the primal and dual solutions.

Now Problem 5(e) probably makes more sense — we've said there's a close relationship, but knowing the primal and dual answers don't tell you the answer to the other. But simplex seems to solve both at the same time (where knowing the answer to the primal gives you one for the dual); so how do those reconcile?

§18.2 The ellipsoid algorithm

Simplex isn't polynomial-time; so for a long time, there was the question of whether polynomial-time algorithms exist. We'll give the big picture of the ellipsoid algorithm, developed in 1979 by Khachiyan. This doesn't focus on optimization so much as finding a feasible point in a polytope. This seems like an easier problem, but we've discussed that it's not; if we can find feasible points in arbitrary linear programs, then we can solve arbitrary linear programs (you take the primal and dual, combine them — so now you're looking for a primal point and a dual point — and you add the one additional equality constraint $yb = cx$. Then any feasible point in the combined LP is going to be optimal for the original primal and dual).

§18.3 Hunting lions

One way to think about ellipsoid is it's really just a generalization of binary search. There's an article about how different branches of mathematics hunt lions in the desert. One is the Bolzano–Weirstrass method, where you have a great big desert and there's a lion. You build a fence acrsos the desert and know the lion is on one side or the other, and ask where it is. When you know this, you build a fence on that side. You keep on splitting the desert in half, and eventually the fencing gets so small that the lion is trapped in a cage.

This is basically what ellipsoid does — it's hunting for a feasible point by doing binary search in a high-dimensional space to slowly hone in on the region where the feasible point is.

In binary search, we take a number and ask whether it's bigger or smaller. In a higher dimension, the natural analog is which side of a given hyperplane the point is on. We're not looking for a specific point, but any old feasible point. So:

> **Question 18.1.** Given a hyperplane, is there a side with no feasible point?

This is the side we need to rule out. If both sides have feasible points then it doesn't matter which side we go to, but if one doesn't then we need to not go there.

In the main loop of the ellipsoid algorithm:

The algorithm offers a candidate point. If it's feasible, we're done; so we only have to worry about what happens if it's not feasible.

If it's not feasible, how do we *know* it's not feasible? There's going to be some constraint that it doesn't satisfy.

Definition 18.2. We call that constraint a separating hyperplane.

That constraint separates the query point being offered from the *entire* feasible region, so we know we need to look on the other side of the hyperplane.

§18.4 Ellipsoids

If we did this literally, eventually we would find all the constraints, and what use would that be? We'd just have the polytope we started with.

But Khachiyan's insight was to work with structures that are less complicated than polytopes. Polytopes are all kinds of weird complicated shapes. But what geometric objects are kind of like a polytope (in that they contain space), but you can ask whether things are inside or outside of it. Any thoughts on what such a shape should be? It'll be an *ellipsoid*.

That's what the algorithm does. We've got our polytope, but that's way too complicated to think about. So what we'll do instead is we maintain a big ellipsoid around that space. And we're going to ask about the *center* of the ellipsoid — is that a feasible point? If it's not, there's going to be some violating constraint demonstrating that this point is not feasible. We'll simplify things by moving that violating constraint over to pass through the center; so now we have half the ellipsoid that contains the feasible region, and half that doesn't intersect the feasible region.

But that's only half the ellipsoid, which is more complicated than an actual ellipsoid. So what do we do? We draw a *new* ellipsoid that contains all the half of the ellipsoid that contained the feasible region (so we still contain the feasible region), but it leaves out a lot of the rest — it's *smaller* (lower-volume).

This is how you measure progress. We start with a gigantic ellipsoid, and keep using these queries to keep shrinking the ellipsoid. We go until the ellipsoid is too small to contain the entire feasible region. And that creates a contradiction — by definition the ellipsoid always contains the feasible region. So the algorithm has to stop before you get to that point, and the only way this happens is if you found a feasible point.

That's the big picture; we're going to fill in some of the details, but we can't fill in all of them (it takes a big book).

§18.5 Finding a smaller ellipsoid

The key insight is the idea that you can find a smaller ellipsoid that still contains half of the bigger one.

Definition 18.3. An ellipsoid is defined by a radius matrix D and a center z; we define

$$E(D, z) = \{x \mid (x - z)^\intercal D^{-1}(x - z) \leq 1\},$$

where D is positive semidefinite and invertible (equivalently, $D = BB^\intercal$ for invertible B).

This is the natural generalization of $x^2 + y^2 \leq 1$ (giving you a circle).

Example 18.4

The ellipsoid $E(I, Z)$ is a unit sphere. In general, $E(D, Z)$ is an affine transformation of the unit sphere that takes $x \mapsto Bx + z$ (you transform your space by stretching and rotating, and then shift the center of the ellipse somewhere).

Given that every ellipsoid is just an affine transformation of a sphere, if we want to show that you can always find a smaller ellipsoid that covers half of an ellipsoid, we can focus on a sphere — we can say we're looking for a smaller ellipsoid to cover half of this weird ellipsoid, so we can undo the transformation of the big ellipsoid to turn it back into the sphere, find an ellipsoid that covers half the sphere, and then redo the transformation back to the original coordinate system. What we care about is the ratio of volumes (we want the volume of the new ellipsoid to be less than the old one). And ratio of volumes is preserved under the linear transformation. So it's enough to show how to construct a smaller ellipsoid to contain half a sphere.

This is mainly going to be a proof by picture. We've got a sphere, and we're going to assume that the separating hyperplane is the vertical (the picture is in 2D, but in reality there are many more dimensions we can't see on the board).

If we want to cover the right half, you can ask, where should you move things around? We want to drop the stuff on the left, so it makes sense to move the center of the ellipse to the right. On the other hand, all the other directions are symmetric, so there's no reason to move the center up or down. So we're going to move it by some distance ε to the right.

And now we need our new ellipsoid to still contain the appropriate half of our starting sphere.

We've drawn this picture to convey intuitively what we want to do — we're going to squash the ellipsoid in the horizontal direction. But if we just squashed the sphere, then we would lose the topmost point. So in order to counter the horizontal squashing, we're goig to have to stretch the ellipse a bit vertically to recover what was lost. This is a bit worrisome, because that might grow the volume beyond what we gained by squashing in the first direction. But let's figure out the math.

What are the sort of constraints that need to be satisfied by the new ellipse? Well, let's first write its formula — the formula for our ellipsoid is

$$\frac{(x_1-\varepsilon)^2}{d_1}+\sum_{i>1}\frac{x_i^2}{d_i}\le 1.$$

(Our center is still going to be at 0 in all the other dimensions, but we've moved the center over.) And this has to contain the right half of the sphere.

(The ellipsoid is axis-aligned, so we actually have a diagonal matrix.)

In 2 dimensions, the points that are important are the top point and bottom point — they're the first ones we're going to lose if we shrink the sphere too much. The rightmost point also matters a lot.

So these three points clearly matter; let's plug them in and see what constraints this imposes on our d_i.

If we look at the rightmost point, this tells us we need

$$d_1^{-1}(1-\varepsilon)^2=1.$$

That tells us we should set $d_1=(1-\varepsilon)^2$. (We actually have an inequality, but we're trying to make the ellipsoid as small as possible, so we should choose values that make this inequality tight.)

Now let's look at other constraints, e.g., $(0,1,0,\ldots)$ (there's one of these for each of our dimensions). We've already got d_1, but because of the shifted center, even though the first coordinate is 0, there's still going to

be a contribution of the first coordinate. So we get

$$\frac{\varepsilon^2}{(1-\varepsilon)^2} + d_2^{-1} = 1$$

(if this constraint is tight). This tells us we should have

$$d_2^{-1} = 1 - \frac{\varepsilon^2}{(1-\varepsilon)^2} \approx 1 - \varepsilon^2.$$

(The $(1-\varepsilon)^2$ term in the denominator does not dramatically change things.) This is true in all the dimensions except the first, so this tells us all the diagonal elements — the stretch and squish in each direction.

Then how do we compute the overall change in volume? We just multiply all the factors — the volume ratio is just the product of all the d_i terms, except that D is actually the *square* of the transformation matrix, so we actually need to take a square root. So we get

$$\text{volume ratio} = \sqrt{\prod d_i}.$$

When we plug everything in, $d_1 < 1$, and $d_2^{-1} < 1$. So we get

$$\text{volume ratio} = \sqrt{\frac{(1-\varepsilon)^2}{(1-\varepsilon^2)^n}} = \frac{(1-\varepsilon)}{(1-\varepsilon^2)^{n/2}}$$

(really we have $n-1$ in the denominator, but it doesn't matter).

How does this compare to 1? It depends on ε. But we get to pick ε. On one hand, a big ε is good because it makes the numerator small; but that also shows up in the denominator, and that might cancel out the improvement from the numerator. So we have to balance these things out. If we set $\varepsilon \approx 1/n$, then what happens? In the numerator, we have $1 - \frac{1}{n}$. Meanwhile, in the denominator, we have

$$\left(1 - \frac{1}{n^2}\right)^{n/2}.$$

When you do this, you basically multiply these things together, and get roughly $1 - \frac{1}{2n}$ in the denominator. So we get

$$\frac{1-1/n}{1-1/2n} = 1 - \Omega(1/n).$$

And this is good. So with all of this work, we're able to reduce the volume of the ellipsoid by a $1 - 1/n$ factor. This is not much, but it's *enough*.

> **Remark 18.5.** We defined our ellipsoid by the constraints coming from the 'corner-points,' which the ellipsoid has to contain. We've glossed over the fact that this is *sufficient*, in that the ellipsoid we defined really does contain the rest of the sphere as well; but this is true.

So the summary now is:

> **Lemma 18.6**
>
> Given an ellipsoid and a separating hyperplane through the center, we can construct an ellipsoid with about $1 - 1/2n$ of the volume that contains the 'good' half of the original ellipsoid.

So if we go back to the plan outlined, we've got a polytope; you make a query and get a separating hyperplane; and then you construct a new ellipsoid that still contains the old hyerplane, but is smaller by a $1 - 1/2n$ factor. Note that this means after n iterations, you shrink the volume by $(1 - 1/2n)^n \approx 1/2$. So you can get a constant-factor shrink in the volume of the ellipsoid in polynomially many steps.

This is very promising, except that we don't have a good sense of our starting and ending ellipsoid — if you have to start with an infinitely large ellipsoid or finish with an infinitely small one, then this wouldn't matter. So how big an ellipsoid do we have to start with, and how small a polytope can we get?

§18.6 Starting ellipsoid size

On the big side, how big does the ellipsoid have to be? Is there any reason to believe the ellipsoid has some finite size? The LP could be unbounded, and that would be worrisome. We can transform into standard form, at which point the *objective value* of the LP becomes bounded; but it might still stretch out in other directions.

But for now, let's start by assuming we have a bounded polytope. How big can it be?

In a much earlier lecture, we proved that all the vertices have coordinates with $n^{O(1)}$ bits (vertices are the result of multiplying an inverse of the constraint matrix by something, and in the inverse of the constraint matrix, all the numbers have polynomial number of bits). Then we can draw a sphere with radius having $n^{O(1)}$ bits, to contain all such points. So this gives us a bound that we can start with — we can start with a sphere whose radius requires a polynomial number of bits to express, i.e., the radius is of the form $2^{\mathsf{poly}(n)}$. And it's a n-dimensional sphere, so that means the volume is also $2^{n^{O(1)}}$ (it's the radius to the nth power, times some weird constant). That's a scary big number, but we're also benefitting from a geometric rate of progress in our ellipsoid algorithm — so if the radius is $2^{n^{O(1)}}$, then it only takes $n^{O(1)}$ halving steps of the volume to reduce its volume to 2 (or any other constant we want).

That's promising; 2 seems like a small volume. But is it small enough? Well, the answer is not really.

§18.7 Ending ellipsoid size

> **Question 18.7.** How small can an ellipsoid be while still containing the entire feasible set?

Now we run into a problem. The smallest feasible set is a single point, which has volume 0. More generally, the feasible set might not be full-dimensional, and that implies it's going to have volume 0. That doesn't work — there's no way we can shrink the ellipsoid enough to have volume 0.

But we've said earlier that all the interesting points (vertices) have polynomially many bits in them. So you can imagine marking those points in your high-dimensional space, and they form a *lattice* — evenly spaced points with the distance between them depending on how many bits you have. These are all the points that might be interesting. And we're worried that we have some low-dimensional feasible set, e.g., a line segment. This is defined by our constraints — if we think of our constraints as having the form $Ax \leq b$, some put together might define a line like this. We need a nonzero volume in order to be able to terminate our ellipsoid algorithm. How do we get it? Well, we change this set of constraints into $Ax \leq b + \varepsilon$. What this has the effect of doing is blowing up a little balloon around your 0-dimensional space. Every previously feasible point is now an *interior* point, because it's no longer tight on any of the constraints. And that implies this new polytope is full-dimensional. So we no longer have the zero volume problem to worry about.

We might have to worry about a really *tiny* volume; how do we control that? Well, we only use a polynomial number of bits in our ε; we choose ε to be small enough to ensure the balloon doesn't get big enough to contain any lattice points it didn't already contain. (The lattice points are spaced out, so if we only grow the balloon by a little bit, we don't grow to contain any new lattice points.) So any lattice point we find in the balloon is actually going to be a feasible point for the original polytope.

So we're actually going to look for a feasible point in the *expanded* polytope. There's a small problem — ε has to be smaller than all the numbers in A in order to avoid absorbing any new lattice points, so working with this smaller ε actually creates a whole new lattice of closer spacing. But you are full-dimensional.

And the volume of the new polytope has a polynomial number of bits. Why? This is because it's a full-dinensional polytope, and it contains a bunch of vertices whose coordinates involve only a polynomial number of bits. And it requires some math to prove, but if you find $n+1$ points and draw a simplex around these points, you can show that because all the coordinates of these points are reasonably large (only $2^{-\mathsf{poly}(n)}$), the volume of what's contained between these vertices is also of the same scale. (You can't make an infinitely low-volume shape out of points with such coordinates.)

So you have a thing with volume $2^{-n^{O(1)}}$, which means it only takes you $n^{O(1)}$ steps to get down from your constant volume to smaller than the volume of the polytope you just built.

Student Question. *Why does adding ε give you a full-dimensional space?*

Answer. It's because only full-dimensional polytopes can have interiors. If we had a feasible point for the original, it might have been tight on some constraints, but when we add ε it's not tight on anything, so it's an interior algorithm.

§18.8 Runtime

The ellipsoid algorithm is polynomial — but that's thanks to these arguments about the sizes of the numbers. That means it's only *weakly* polynomial — its runtime depends on the sizes of the numbers you're working with. But it does meet the requirement of polynomiality in terms of input size. The question of whether there's a *strongly* polynomial algorithm is still open. (In fixed dimensions, there are strongly polynomial algorithms using randomization.)

§18.9 Separation oracles

There's one other amazing thing that went by without noticing. We talked about a loop where we take a point and get a feasible hyperplane. So at each step, all we need is for someone to give me a violating constraint. This means I, running the algorithm, don't need to know the constraints. In fact, the polytope could be defined by an exponential number of constraints I don't know about. As long as there's a separation oracle — a procedure that gives me a violated constraint — I can run the ellipsoid algorithm.

In other words, the ellipsoid doesn't need the polytope. It only needs a *separation oracle* that gives *one* violated constraint for any feasible point. And that works even if there's a tremendous number of constraints and you never actually want to write them down.

This means ellipsoid works for linear programs that are too big to write down, as long as you get a separation oracle! This is really useful in theoretical computer science.

So a separation oracle allows you to use ellipsoid to solve feasibility, which we know also allows you to solve optimization. And just to complete the loop, theoreticians have proven that if you can optimize over a polytope, then even if there are too many constraints to write down, you can build a separation oracle for that polytope. So these three problems are all equivalent for linear inequalities.

There are many accomplishments in theoretical computer science that rely on taking some hard problem and developing a separation oracle, because that lets you optimize.

Student Question. *Doesn't the optimization to feasibility reduction blow up the number of vertices (since taking the dual means creating a variable for each constraint)?*

Answer. Ellipsoid is perfectly happy to have exponentially many constraints, but you have to have a

polynomial number of variables. And if the primal has too many constraints, then the dual is going to have lots of variables, and that could be a problem.

So the standard reduction from optimization to feasibility could create too many variabes, if you started out with an exponential number of constraints. To address this, we use a different reduction from optimization to feasibility if that issue arises.

If we want to be able to reduce optimization to feasibility, suppose we start wiht $Ax = b$ and $x \geq 0$, and we want to optimize $\min cx$. If we start with that, how do we use feasibility? Well, we literally do a binary search on the value of the optimum – we turn this into $Ax = b$, $x \geq 0$, $cx \geq$ target, and ask the ellipsoid algorithm if that's feasible. Depending on the answer we make a bigger target, or a smaller target. And all nubmers have a polynomial number of bits, so we only need accuracy at a polynomial number of bits to get an optimum.

Student Question. *Once ellipsoid is done, how do you get a feasible point in the actual polytope?*

Answer. You use an argument similar to what we used to transform an interior point into a vertex — you can use a similar technique to get to an original lattice point, as long as you're close enough that there are no other vertices that are not optimum but closer to you than the optimum is. You basically just have to get beyond the second-best vertex. And we showed you can always round to a point that's no worse in objective than the starting point. So if we start at a point that is better than the second-best vertex, then we have to go to the optimum vertex.

Remark 18.8. Note that the fact $d_1 < 1$ represents that we squashed the ellipsoid in the direction of the separating hyperplane. And $d_2 > 1$, which represents the fact that we had to stretch in the other directions in order to contain the other tight points.

§18.10 Interior point

Now we're going to talk about the interior point algorithm, which is going to make the ellipsoid algorithm look very formal; we'll really just give the flavor.

Why did people develop this? The best implementation of ellipsoid costs n^6, so it doesn't have much impact in practice (but ideas have — the idea that you don't have to look at the constraints has worked its way in terms of *column generation*, where if you have a really big linear program you only give the solver the constraints you think are important, and if it gives you an infeasible point then you put in more constraints).

But people wanted practical polynomial time algorithms, and eventually came up with this.

§18.11 Physics motivation

Simplex drops a ball, and it rolls down the sides of your well. The problem is that every time it hits a vertex, there's a mess that happens — it bangs into another wall and it ricochets and a bunch of physics happens. So how can you do something simplex-like but avoid colliding with all these corners? The answer is you use an iron ball and put a magnet behind each of the walls. What is the effect of this? Well, as the ball gets closer and closer to the wall, it's actually going to be repelled by the wall. So instead of rolling down and collidiing with each of these points, it's going to drift its way down, a bit away from the walls. It'll be dragged down by gravity. Eventually it'll end up close to the optimum — not quite at the optimum because of the magnets, but as long as you can get close enough ($1/2^{\mathsf{poly}(n)}$) that's good enough (by the same rounding thing mentioned earlier).

§18.12 A potential function

This technique of putting in magnets was originally broadly known as *barrier methods*. In the 1980s it was first shown how to make this practical. The idea is to develop some math that does what these magnets do.

Instead of trying to optimize a linear objective function facing all these tight linear constraints, instead you turn all the constraints into a *(nonlinear) potential function*. And now that you have this potential function, you do *unconstrained* optimization. And if you have a nonlinear function and want to optimize it, what do you use? You use gradient descent (derivatives).

This is a lovely idea, and all the devil is in the details (which we won't discuss).

But assuming standard form, the constraints all have the form $x \geq 0$ (and there are the equality constraints, but those don't matter — we can just work in the subspace they define — and it's just the $x \geq 0$ constraints that make things hard). So we define potential

$$G_\mu(x) = cx - \mu \sum \log x_j$$

where μ is some 'magnetic constant' (and the reason for the logs is just that it works). In global terms, cx is still in the objective function, so we want to minimize that still. But working against us is the fact that if any x_j gets very close to 0, then the log term becomes very negative, which means this quantity adds substantially to the potential function. So this potential function is basically saying to minimize cx but not let any of these x_js get too close to 0, which is exactly what we want.

Once we have defined this potential function, now you think about using gradient descent. You have

$$\nabla G_\mu(x) = c - \sum \frac{\mu}{x_j}$$

(this is one reason you use log — it makes for a very nice gradient). And the gradient tells you a direction you can move in to improve your potential. As long as you don't move too far, the gradient is a good approximation, and moving in that direction means you're improving the potential.

This requires rigorous analysis, but you can show there's a suitable step size, which oddly enough shares the $1/n$ behavior from ellipsoid (when in the right units). And you can show this gives you sufficient improvement step by step such tha tafter only polynomially many steps, you're close to the optimum of the potential function, and therefore also the objective. You also have to choose μ appropriately, and it turns out $\mu \approx 1/n$ is good.

§18.13 The central path

This is how it was originally developed — make this potential function and set μ appropriately. More modern interior point algorithms have developed more insight to what's going on. They observed that given any particular μ, the quantity $G_\mu(x)$ has an optimum somewhere. If $\mu = 0$, then the optimum is at the optimum of the polytope (the barrier is gone). If μ is very large, then it doesn't matter what the objective function is (the point will just stay in the middle of the polytope, as far from the constraints that it can be). Varying μ from ∞ to 0 traces out a curve in the polytope — it starts at the center, and it ends at OPT. This is called the *central path* for this optimization problem.

So you have your polytope and start at the center with $\mu = \infty$, and as you vary μ you move down to the optimum. Modern interior point algorithms are *path following* or *path correcting*. Assuming we've gotten somewhere on the central path, we approximate the central path where we are by a straight line; we move some distance along the central line, which is kind of like moving along the central path. Path following algorithms go from there; path correcting go from there onto the actual central path. And then you repeat. SO we basically discretize the central path into a bunch of lines, and show that you can make these lines long enough that you only have to move polynomially many times.

Now these sorts of techniques are also used for max-flow and other combinatorial optimization problems (there's a 120-page paper on interior point algorithms for max-flow, that achieves near-linear time). But these are only approximation algorithms — there's a gap in the end that you have to clean up by some rounding technique.

That's all we'll say, to not get into gradient descent and continuous optimization.

Next week will be approximation algorithms. Friday will not have a class.

§19 October 28, 2024

§19.1 Dealing with NP-hard problems

We've been developing polynomial-time algorithms for increasingly hard problems. Last week we looked at linear programming, which got quite mathy, because those were really hard. Now we're going to push further to problems where we don't think it's *possible* to have polynomial-time algorithm; what do theoreticians do in that case? Theory has developed this whole notion of NP-hard problems — these are problems for which there is no polynomial-time algorithm, unless $\mathsf{P} = \mathsf{NP}$. When you're faced with this kind of obstacle, there's a few options you can consider.

One is to develop non-polynomial-time algorithms with provable bounds. For example, there's a lot of work that says, of course it takes exponential time to solve SAT, but what's the base of the exponent? There's a lot of cool work on pushing that base down. (If you're not a theoretician, you might not care about provable runtime and just want something that works in practice, but that's out of scope for this class.)

Another approach is to consider special cases — if we only consider degree-3 graphs with diameter at most 17 and it's a Tuesday, maybe we can get a polynomial-time algorithm. This can have particular impact if you identify a particular special case that comes up often in practice.

Another approach people often take is to assume random inputs, and discuss the *expected* runtime. If you're looking at the Hamiltonian path problem, that's NP-hard; but Hamiltonian path on a random graph is polynomial-time solvable in expectation. The problem is that the real world is not random, but is out to get you; so whatever your distributional assumptions are, they might not be met in the real world. Even just among theoreticians, agreeing on the right distribution is difficult. When talking about a distribution, you're saying some problems are easy and form a majority, and the hard ones are rare; so you're kind of doing special cases again.

But we're not going to do any of those things. Instead, we're going to preserve the focus on polynomial-time and on solving all instances, but we're going to settle for non-optimal answers. That's going to be our focus this week.

§19.2 Setup

§19.2.1 Optimization problems

To get some definitions on the board (to be clear what we're talking about), so far we've been talking a lot about optimization problems, which consist of a set of input instsances $\mathcal{I}$, each of which comes with a set of feasible solutions $\mathcal{S}(I)$ to which we assign values — there's a value function $f: \mathcal{S}(I) \to \mathbb{R}$ (that maps every solution to every instance into the integers or reals or some other ordered set). We've been looking at both maximization and minimization problems. And we'll refer to the optimum solution and its value as $\mathsf{OPT}(\mathcal{I})$ (we're overloading this notation — in some cases it'll refer to the optimum solution and in others its value, but which will be clear from context).

Example 19.1 (Bin-packing)

In the bin-packing problem (which you solve every time you move out of a dorm), you have a bunch of things that you need to pack into boxes of a certain fixed size; each thing takes up some space in a box. Your goal is to figure out the smallest number of boxes that you can use to pack all the items.

Here every instance I is a set of item-sizes $s_1, \ldots, s_n$; and your goal is to fit all of them into bins of size 1 (you have to assign each item to a particular bin, and the total size of items assigned to every particular bin cannot exceed 1). And your goal is to use the minimum number of bins.

(This is going to be a very refreshing lecture because we'll get introduced to lots of fun algorithmic problems and get digestible solutions, rather than doing a bunch of math.)

We're going to make some technical assumptions. First, we assume all our inputs and the range of f are integers or rationals (because you can't represent reals on a computer), with polynomial sizes (alternatively, we'll talk about the runtime in terms of the input size, which includes paying attention to the sizes of the numbers placed in the input). We're going to look for algorithms that are polynomial-time in the bit sizes (we're not going to think about strongly polynomial algorithms in this domain).

§19.2.2 NP-hardness

We mentioned NP-hardness earlier. When you're introduced to NP-hardness in complexity theory, it's generally when talking about decision problems (yes/no). We simply say that an optimization problem is NP-hard if some NP-hard decision problem can be reduced to it. One obvious example of a problem that may be NP-hard and can be reduced to this optimization problem is to ask whether $\mathsf{OPT}(I) < k$ — if you can optimize then you can answer decision problems of this form, and this is a decision problem so falls into the standard work on NP-completeness. But it doesn't have to be this one — we allow more general Turing reducibility. We're not going to do any reductions in this class; Prof. Karger is just going to tell us certain problems are NP-hard, and we'll believe him because of the long history of trust that has developed between us.

§19.3 Absolute approximation algorithms

The theoretician's definition of an approximation algorithm is simply an algorithm that gives a feasible solution — any algorithm that gives a feasible solution is an approximation algorithm. Of course, the challenge is to decide what makes a *good* approximation algorithm. Put in another way, what's the proper definition of having a solution that is *close* to the optimal solution?

For starters, let's take one of the most obvious definitions, and talk about *absolute* approximations.

Definition 19.2. An algorithm $\mathcal{A}$ is a k-absolute approximation algorithm if for any input I, we have

$$|\mathcal{A}(I) - \mathsf{OPT}(I)| \leq k.$$

§19.3.1 Planar graph coloring

Example 19.3 (Graph coloring)

You're given a graph as input, and you want to color each vertex such that no two neighbors have the same color. Your goal is to minimize the number of colors.

Every theoretician's paper that talks about graph coloring says it has practical applications for register allocations — in compilers, you think of the values as the things you want to color, and the registers as the colors; and the question is whether you can fit all the things that will be used at the same time into your registers. If you assign two values that'll be needed at the same time (represented by an edge) to the same register, then you have a problem. But that's not why we study graph coloring; it's because it's a beautiful problem with lots of interesting and deep theory.

This problem is NP-hard. In fact, it's *very* NP-hard, in ways we'll talk about later (it's very hard to approximate). However, there's the special case of *planar* graph coloring, which is easier to solve.

Even for planar graphs, it turns out to be NP-hard to decide if the graph is 3-colorable — some graphs are and some aren't, and telling them apart is hard. (Deciding whether a graph is 2-colorable can be done in polynomial time, and it's not difficult.)

On the other hand, we know every planar graph is 4-colorable (according to the planar map algorithm). Actually finding that 4-coloring is hard (and involves a bunch of math). But it's actually very easy to 5-color. Since it's easy to 5-color and you certainly need 1 color, that means we have a 4-absolute approximation — we are coloring every single graph using a number of colors that's at most four more than the optimum.

> **Remark 19.4.** In this class, we'll want our algorithms to be constructive (to actually find an optimum, not just the optimum value).

In fact, we can do a bit better. Deciding whether a graph is 2-colorable is easy. You basically force it — you color one vertex red, then all its neighbors have to be blue, and all their neighbors have to be red, and so on. You keep going, and eventually you'll fill an entire component of the graph, and move on to the next. The point is that the existence of a solution tells you what exactly the solution has to be.

So 2-coloring is easy (and 1-colorable graphs are also easy to detect — just the graph with no edges). And 5-coloring can be done greedily. So you only need to use the 5-coloring algorithm on graphs that are 3-colorable or 4-colorable. So that means you actually have a 2-absolute approximation.

There's a similar result for 'edge-coloring' — here, instead of coloring the vertices, you have to color the edges so that no two edges sharing an endpoint have the same color. With edge-coloring, there's a trivial lower bound — if there's a vertex of degree Δ, then all its incident edges have to have different colors. Meanwhile, there's a famous theorem of Vizing:

> **Theorem 19.5** (Vizing)
> We always have $\mathsf{OPT} = \Delta$ or $\Delta + 1$.

This theorem is constructive (and the construction can be done in polynomial time). So this actually gives you a 1-absolute approximation.

§19.3.2 The scaling issue

So we're on a roll — there's two interesting NP-hard problems, and we've quickly dispatched them with very good approximation algorithms.

The problem is that almost no interesting problems have good absolute approximation algorithms. And there's a reason for this. Really, additive approximations only tend to show up when OPT is bounded by a constant (i.e., $\mathsf{OPT} = O(1)$), which was the case for the problems we've just looked at. You can often prove impossibility via a 'scaling' or 'amplification' argument; Prof. Karger will give us one of those right now, and use it to introduce another problem.

Example 19.6 (Knapsack problem)

Imagine you walk into a store and want to steal as much as you can, but there's only a certain amount of space under your sweater. You want to collect as much stuff that will fit under your sweater so that you can walk out the store without paying.

To formalize this mathematically, we can think about items with sizes s_i and profits p_i, and a knapsack of size B. And you want to fit the maximum profit (without reusing items — you only get to take each item once) while fitting within the size. In other words, you want $\max\{\sum_{i\in Y} p_i \mid \sum_{i\in Y} s_i \le B\}$, where Y is what we put into the knapsack.

We can assume without loss of generality that these numbers are all integers (otherwise we can clear their denominators). Now that we're in approximation algorithms, we'll tolerate very large polynomials; so we won't really care whether we have $(\log n)$-bit numbers or n-bit numbers.

Claim 19.7 — Suppose we're able to find a k-approximation to the knapsack algorithm — whatever the profits in sizes are, we can find a solution that's within an additive k of the optimum. Then we can use this to solve the knapsack problem exactly.

(This would be a problem because knapsack is a NP-hard problem.)

Proof. The idea is that if we scale up the problem, then this additive error becomes so small that your solution has to be exact.

Specifically, all we do is we multiply all our profits by $k+1$ (where we have an additive solution within k of OPT). Then any previously non-optimal solution now has value at least $k+1$ greater than the new OPT (since multiplying profits by $k+1$ multiplies the value of every solution by $k+1$, and since we're working with integers, every non-optimal solution originally had value at least 1 greater than OPT, so now it's at least $k+1$ greater).

So a k-absolute approximate solution to the scaled problem is actually an optimal solution for the original (and multiplying by $k+1$ doesn't change the size of the numbers significantly, so it's still polynomial time). □

It's usually easy to see how to scale when working with problems that have numbers, but you can also scale *structures*.

Example 19.8 (Independent set)

Given a graph G, find a maximum independent set of vertices — meaning that no two vertices are adjacent.

This is NP-hard. Its complement is the clique problem, where we want a collection of vertices all connected to each other; they're the same problem (swapping non-edges with edges).

Claim 19.9 — Suppose we have a k-absolute approximation algorithm for the independent set problem (meaning that $\mathcal{A}(I) \ge \mathsf{OPT}(I) - k$), then we can get an exact algorithm.

We're again going to do this by scaling. But now we don't have any numbers; so what do we multiply?

Proof. Make $k+1$ copies of the graph (with no copies between them). Then any independent set in the original graph turns into a replicated independent set in the copied graph, with $k+1$ times the size. So the optimum size of an independent set gets multiplied by $k+1$,

And we can certainly find a set of size $(k+1)\mathsf{OPT}(G) - k$ in the replicated graph. How do we find an optimal independent set in the original graph? The *average* number of vertices per graph in this solution is

$$\frac{(k+1)\mathsf{OPT}(G) - k}{k+1} = \mathsf{OPT}(G) - \frac{k}{k+1} > \mathsf{OPT}(G) - 1.$$

So if the average number of vertices is larger than $\mathsf{OPT}(G) - 1$, then some graph has at least the average. But it has to have an integer number of vertices. So some graph has an integer number of vertices greater than $\mathsf{OPT}(G) - 1$, and this means it has $\mathsf{OPT}(G)$ vertices. □

So even when there aren't numbers, you still get stuck without additive approximations.

That's all we're going to say about absolute (or additive) approximations, because they don't appear often; we're going to have to back up to a weaker notion of approximation, which is what really dominates.

§19.4 Relative approximation

Definition 19.10. An α-optimum solution is a solution of value between $\frac{1}{\alpha}$ and α times OPT.

This might seem like a weird definition, but what it really means is that if $\alpha > 01$, then when you're talking about maximization problems you'll use $\frac{1}{\alpha}$, and for minimization problems you'll achieve α. But some algorithmicists like talking about $\alpha < 1$, so these will flip (some people talk about a $\frac{1}{2}$-approximation and 2-approximation). But there's no ambiguity, since you can only be on one side of OPT (you can never do better than OPT).

Definition 19.11. We say an algorithm has approximation ratio α if it yields an α-approximation on every input; we also call it an α-approximation algorithm.

The whole rest of this unit is going to be about developing these α-approximation algorithms, trying to make α as close as possible to 1.

§19.5 Proving something is an α-approximation

The issue is that the definition talks about how close your solution is to optimal. But we can't *find* the optimal solution (that's why we're doing approximation), so we can't lift it up to compare to our solution. This means we need to use lower or upper bounds on OPT. If we have a minimization problem, we find a lower bound which is smaller than OPT, and we compare how we did to that lower bound — if we're within a ratio of α of the lower bound, then we're within a ratio of α of the actual minimum (which is between us and the lower bound) as well.

There's two bits of cleverness — coming up with the algorithm, and coming up with the lower bound that lets you prove the algorithm is doing well. Often these go together — giving a lower bound can give you a direction for an algorithm that achieves (close to) that lower bound.

§19.6 Greedy algorithms

For the rest of the week, we're going to see a bunch of general techniques for developing approximation algorithms. The first is *greedy algorithms*. The idea is that often you build a solution in a series of steps, and a greedy algorithm just takes what looks like the best step at each opportunity — you just do the apparently best thing at each step.

It's pretty obvious what this is, usually; the hard part is proving that it gives you some kind of useful approximation, which requires coming up with a lower bound that you can compare to what you did.

§19.7 Max-Cut

To get started, we'll look at the max-cut problem. This has been around for a long time, but took on real importance when Goemans and Williamson came up with a stupendous approximation algorithm (leading to a whole branch of approximation algorithms). We won't go over that because it requires randomization. But we will talk about a simple algorithm.

Example 19.12 (Max-Cut)

Given a graph, find a cut of the graph that has the maximum number of edges crossing the cut.

(Here we won't worry about a source and sink; we just want to break the graph into two pieces.)

Student Question. *Do the two pieces have to be connected?*

Answer. No. In fact, it's a good thing to have the pieces be disconnected, because that means you're not wasting edges inside the two pieces.

What would be a greedy algorithm? Imagine placing one vertex at a time. The first vertex goes on some side (it doesn't matter which). But as we start placing other vertices (imagine we add them in a fixed order), which side should we put them on? We can just greedily grow the cut — each time, we place the vertex on the side that increases the value of the cut more. The vertex will have some neighbors on each side, and we want to put it on the side *opposite* the majority of its neighbors.

Algorithm 19.13

Add one vertex at a time, to the side where it currently has fewer neighbors.

Some of its neighbors haven't been added yet, so we don't know what'll happen with them, but we can be greedy about the vertices which have already been added.

How do we analyze this natural greedy algorithm? This is a maximization problem, so one thing we need is some kind of upper boun. What's an upper bound on the maximum possible value of a cut? Let's be lazy; a very lazy upper bound is m (the number of edges). And let's compare how we do to that.

There's an intuition here — what we said is you add a vertex to the side where it has more of its edges on the other side. So if we look at just the edges being decided by this addition, we're cutting at least half of them — when we place each vertex, we put it opposite most of its neighbors, which means at least half of the edges being determined (i.e., whether they're cut or not) in this moment are actually cut.

In other words, when we add a vertex, it determines some edges — those going to the already added vertices. So we can think of a series of rounds, where in each round, some of our edges are determined to be either cut or not cut. And over the course of the whole algorithm, all edges will be determined to be cut or not cut. And in this round, at least half these edges are cut; and since all edges get determined at some point, this means over all the steps, at least $\frac{m}{2}$ edges are cut.

So this gives a nice, simple 2-approximation.

§19.8 Set cover

Example 19.14 (Set cover)

You have a collection of items, and a collection of subsets of those items. You want to find a subcollection of these subsets that 'covers' all the items — i.e., their union is all the items. And your goal is to use the fewest possible covering sets.

What would be a natural greedy algorithm for covering all the items with as few sets as possible? You want to take the set that covers the maximum number of previously uncovered items.

Algorithm 19.15

Repeatedly take the set that covers the most uncovered items.

This is pretty natural; the trick is the analysis. In this problem, we're not going to develop a lower bound; instead we're going to compare to OPT, which we'll call k. Suppose you can cover with k sets. Instead of analyzing our solution overall, we'll argue we make a lot of progress as we do this greedy algorithm.

At the start, there are n items to be covered (it doesn't matter how many sets there are). And we're taking a set that covers the most items. We claim we can get a good amount of progress from this set — there has to be a set that covers $\frac{n}{k}$ items (we can cover everything with k sets, so some set must be covering at least $\frac{n}{k}$ algorithms, by averaging).

In other words, of the optimum sets, one must cover at least $\frac{n}{k}$ items in the first round. This implies that the greedy algorithm also covers $\frac{n}{k}$ items in that round.

Then what can we say about the second round? In general, if we have t items, then some set has to cover $\frac{t}{k}$ items — whatever number of items are left, we cover at least a $\frac{1}{k}$-fraction of them. But if we covered $\frac{t}{k}$ items, then afterwards there's $(1-\frac{1}{k})t$ items remaining.

So more generally, after r rounds, there will be at most $(1-\frac{1}{k})^r n$ uncovered items. This means after $O(k \log n)$ rounds (we have $(1-\frac{1}{k})^k \approx e$), there will be less than $\frac{1}{n} \cdot n = 1$ items uncovered; this means there's 0 items uncovered (since the number of uncovered items is an integer).

So whatever the optimum value k was, this greedy algorithm manages with $k \log n$. This means we have an $O(\log n)$-approximation.

There's always the obvious algorithmicists' question, can you do better? Our theory suggests the answer is no (for general set cover problems).

§19.9 Vertex cover

A close relative of set cover is vertex cover — it's actually a special case.

Example 19.16 (Vertex cover)

Given a graph, find the fewest possible vertices that 'cover' all the edges.

Note that in set cover we wanted to cover items by sets; similarly here we want to cover edges by vertices (it's the thing that's in the cover that's the name of the problem). This is a special case of the set cover. What's special about it is that every element appears in (exactly) two sets (since every edge is covered by two of your vertices, and you'll have to take one of them in any cover).

One approach we could use is to specialize the set cover algorithm; that specialization would be to pick the vertex of largest remaining degree.

But it turns out that in this problem, greed is good, and more greed is better — the approach is a *super greedy* approach. Suppose there's some edge that hasn't been covered yet. We know one of its endpoints has to be in the cover; the greediest thing we can do is to take *both* of them.

Algorithm 19.17

Find an uncovered edge, and add *both* the ends to the cover.

We're being greedier than we have to be — greedily covering edges one at a time would mean just taking one vertex. The reason this is better is that when we take a look at this edge, we know one of its endpoints *has* to be in the optimum cover. If we only take one endpoint, we might be taking the wrong one. But if we take both, then we're definitely taking one of the vertices of the optimum cover.

This means each step reduces OPT by 1. (This is what's different from set cover — there when we greedily take a step, it might not be in the optimum, so we might not be reducing OPT.) This means the number of steps is at most OPT. And we're only taking two vertices in each step, so our solution is at most 2OPT. So for this special case, we get a 2-approximation instead of a $O(\log n)$-approximation, just by being 'extra greedy' in what we do.

> **Remark 19.18.** In fact, we can't do any better than this (except infinitestimaly).

§19.10 Scheduling theory

The last problem we'll talk about draws from a broad area called scheduling theory (we're probably all practicioners of this, and maybe this will help us get more sleep).

> **Example 19.19**
>
> Suppose we have m machines which can run n jobs, where the jobs have processing times p_j. Our goal is to assign each job to a machine to finish all the jobs as quickly as possible. In other words, we want to minimize the maximum over machines of the sum of the p_j's of all jobs assigned to that machine (we call this $C_{\max}$, the *maximum completion time*).

So we have fixed processing times, and we want to minimize the maximum time. You could also imagine a continuous version, where a job imposes a certain load on every machine and runs continuously, and you want to minimize the maximum load.

This seems suspiciously similar to the bin-packing problem, and in terms of exact solutions they're the same — that's asking how many machines you need in order to finish within a certain time, while this asks how much time it takes to finish given a certain number of machines. So an exact solution to one would give you an exact solution for the other. But we don't have exact solutions, and the approximation techniques and ratios are quite different.

(In this instance, the machines are all identical; the p_j are indexed just by jobs, not by machines.)

What's an obvious greedy algorithm? We're going to create the solution one step at a time. On an individual basis, what's the smart thing to do at each step?

> **Algorithm 19.20** (Gram's algorithm)
>
> One job at a time, assign the job to the currently least-loaded machine (the one that has the least work on it so far).

This is a very natural algorithm. The insight is in thinking about how to prove that it does well. In particular, what sort of lower bound can we compare this to?

This problem is interesting, because you really need to think about *two* lower bounds in order to get a result.

One natural lower bound is the *average* load $\frac{1}{m}\sum p_j$. Just before we get to the other lower bound, one thing that's important is thinking about how close the lower bound is to the optimum. If we have a lower bound that's way worse than the optimum, then when we prove a ratio to the lower bound, it's going to be

terrible — my solution can't be better than OPT, so if the lower bound is much worse, then we get a really bad ratio.

And this lower bound is very weak. For example, imagine you have just one job; then the average load is $\frac{1}{m}$, but the best we can do is 1. So that's a factor of m off.

So this lower bound is not always good. What do we do? We come up with a different lower bound — the largest job size, i.e., $\max_j p_j$, is also a lower bound on OPT. This is *also* a terrible lower bound, e.g., if we have n equal-sized jobs.

However, it turns out that if you use *both* of these bounds, then things work well — one of these bounds is always good. To prove that, let's consider the final max-load machine — after we've assigned all the jobs, some machine has the maximum load. Why does it have the maximum load? It had some old load ℓ, and then we added some job of size p_j. (And that's the last job you added to the machine.)

Now, why did you add the job to this machine? At this moment, all the machines must have had load at least ℓ. It follows that the average load is at least ℓ, which of course means $\mathsf{OPT} \geq \ell$ (since the average load is a lower bound).

Then we added p_j. But p_j is also a lower bound on OPT, looking at the other bound. So the final load of this machine is

$$C_{\max} = \ell + p_j \leq \mathsf{OPT} + \mathsf{OPT} = 2\mathsf{OPT}.$$

And so we have a 2-approximation.

In fact, we can say a little bit more — in fact, the average load when we finish is at least $\ell + \frac{p_j}{m}$ (since it was ℓ before we added this job in, and we added another job of size p_j). And we end up achieving $\ell + p_j$, which we can rewrite as

$$\left(\ell + \frac{p_j}{m}\right) + p_j\left(1 - \frac{1}{m}\right).$$

But $\ell + \frac{p_j}{m} \leq \mathsf{OPT}$ (by the argument from before), and $p_j \leq \mathsf{OPT}$. This means we actually get a $(1 - \frac{1}{m})$-approximation (which is slightly better when you have a small number of machines).

One thing to mention about this algorithm is that it's *online* — you can run this even if you have to assign jobs as they arrive (you're at the door, a new job arrives, and you put it on a machine). And even so, the maximum load is at most a factor of 2 worse than what you could do knowing all the jobs ahead of time.

And again in this problem, we have no idea what OPT is; we just compared to the natural lower bounds.

This bound is actually tight (for the analysis of this algorithm), and you can prove this by combining the bad cases we discussed for the lower bounds. If you start by schedulign $m(m-1)$ size-1 jobs (the greedy algorithm will spread them out evenly), and then one job of size m (which will land on top of these $n-1$ small jobs), then we'll have $2m-1$ on that machine, and all the others will have $m-1$. What you should have done instead is put the big job on one machines, and all the small jobs on the other.

What this says is you should pay attention to the big jobs first. And you can prove (this might be on the homework) that if you schedule in order of processing time (where you do greedy scheduling with longest-processing-time jobs first), then you get a $\frac{4}{3}$-approximation. We won't dwell on that because we can actually do much better, but we'll need more sophisticated techniques (which we'll discuss later in the week).

§20 October 30, 2024

Today we'll dive deeply into approximation algorithms. Last time we saw the definitions and some obvious approaches. Today we'll talk about some much less obvious approaches.

§20.1 Scheduling theory

We're actually going to pick up where we left off, with the problem of scheduling and jobs.

Example 20.1

Shedule n jobs on m machines to minimize the maximum completion time.

This is just one example of a problem from scheduling theory. Scheduling theory is a whole branch of operations research (getting things out of factories on time, planning people to work efficiently) and we've developed a whole taxonomy of scheduling problems, that can all be addressed using related machinery.

Last time, we looked at a specific instance. But in the general case, you have some collection of m machines. There's many different characterizations of those machines. Lots of problems look at the special case of 1 machine, others look at m identical machines (as we had). Others have m machines of different *speeds* (a machine which runs twice as fast would process every job twice as fast as another machine). Or you might have unrelated machines — depending on the characteristics of the problems and the machine, the processing times could be *anything*. So here you'd have processing times represented as p_{ij} (for job j on machine i).

And that's just for computing. When you get into manufacturing, many jobs need to be processed on *multiple* machines in sequence (you might need one machine to mold something, another to drill holes, and so on). So you might have a *flow shop*, where each job requires a specific amount of work on specific machines in order. Or you might have an *open shop*, where the work involves many machines, but the order doesn't matter (you can attach widget A and B in either order, as long as you attach both).

With so many different machine types, we've come up with abbreviations for everything — m identical machines as P, m related machines as Q, m unrelated machines as R (unfortunately), flow shops as F, and open shops as O.

There are also many objective functions you might want to consider. We looked at the maximum completion time $\max c_j$. But you might also want to look at, say, the *average* completion time, which is basically $\sum c_j$ (up to scaling). Or you might want to look at $\sum u_j$, where u_j are indicator variables for a job being *late* (past its deadline — you can imagine adding a deadline to each job and saying you don't care exactly when the jobs finish, as long as they all finish before their deadlines). Or maybe we want weighted averages $\sum w_j c_j$ or $\sum w_j u_j$ (because maybe some jobs are more important). There's also the flow time $\sum f_j$ (where f_j is the time that a job is in the system).

The third thing you can have is a whole bunch of constraints or additions. These include things like special cases (maybe all the processing times are 1). You might have *release dates* r_j (for each job), and you have to process a job only after its release date. You might say that jobs have deadlines d_j by which they're supposed to finish. You might allow *preemption*, which means you can process a job for a while and then set it aside to do a different job, and then come back and finish processing it later (as long as the total processing time adds up to what the job requires). And then there's *precedence constraints* on the jobs, which say you cannot do job 17 until you've finished job 3 (because there's some dependency between them).

So you've got all these different machine models, all these different objectives, and all these different constraints. That gives you a combinatorial explosion of scheduling problems — you can construct thousands (the constraints can be taken in combination, for example). Some are very easy (linear-time solvable), some are NP-hard, some we don't know. Years ago there was a scheduling problem classifier online, where you could type in a scheduling problem and ask for the state of knowledge. Specifically, it told you which are the hardest scheduling problems we don't know to be NP-complete, or the easiest ones we don't know how to solve — because there are reductions between all these different schedulling problems. If we can't think of a problem for our final projects, there's a very large supply of stuff here.

There's a notation for putting all of this together, where you write down the machine model, the extra constraints, and the objective. Under that notation, we just did P $||$ $C_{\max}$ (the maximum completion time

when you're scheduling one machine). But we'll look at other scheduling problems as well, and it's a rich source of interesting theory problems.

When we did this problem, we had a greedy algorithm. The analysis said that you look at the last job to complete, and it had processing time p_j. It was added to a machine of minimum load ℓ (at that time) — that's why it was added. Then we observed that the average load of all the machines taken together was at least

$$\ell + \frac{p_j}{m} \leq \mathsf{OPT}.$$

We also argued that $p_j \leq \mathsf{OPT}$ (every job has to be scheduled *somewhere*). This told us that

$$C_{\max} = \ell + p_j = \ell + \frac{p_j}{m} + \left(1 - \frac{1}{m}\right) p_j \leq \mathsf{OPT}\left(2 - \frac{1}{m}\right).$$

So we had a $(2 - \frac{1}{m})$-approximation for $\mathrm{P} \,||\, c_{\max}$.

§20.2 The best approximation ratios

This was the last of several approximation algorithms we looked at last time — vertex cover, set cover, and graph coloring. We got various approximation ratios for these different problems.

> **Question 20.2.** Are these the best possible approximation ratios? Could we do better?

You can ask this about specific problems, or you could ask it in general about classes of problems.

> **Question 20.3.** What is the best achievable approximation factor?

Rght now, all we really know is you can't achieve 1 (that'd be an exact solution, and these are NP-hard problems). But theoreticians wrestled with this and came up with answers in two different directions.

Some problems have a lower-bounding constant — there's some constant above (or below) which you can't approximate these problems (assuming $\mathsf{P} \neq \mathsf{NP}$). We call these APX-hard (for 'approximation').

There's a whole complexity theory saying that if you could approximate this problem better than a certain value, then you would be able to solve NP-hard problems.

On the converse side (the positive side), some problems have arbitrarily close approximations. That's what we'll look at next.

Vertex-Cover is APX-hard — we gave a 2-approximation, and it turns out you can't do substantially better. In fact, you shouldn't be able to do any constant better.

Also for set cover, you can't do better than $\log n$. For clique and independent set, these have *terrible* best approximation ratios; we'll look at this on the homework.

§20.3 Approximation schemes

But now we'll look at problems where you can get an arbitrarily good approximation ratio.

> **Definition 20.4.** A polynomial-time approximation scheme for a given problem is a collection of algorithms $\mathcal{A}_\varepsilon$ such that each $\mathcal{A}_\varepsilon$ is a polynomial-time $(1+\varepsilon)$-approximation algorithm.

So however close you want to get, you specify ε, and the approximation scheme supplies you an algorithm that runs in polynomial time and gets a $(1+\varepsilon)$-approximation.

What's a bit weaselly here is that 'polynomial time' doesn't include ε — if we have one algorithm with a runtime of n^{10000/ε^2}, then that's considered an approximation scheme. (David developed an algorithm with this runtime, which is the reason it's mentioned.)

This is slightly unsatisfactory, so you might instead ask for the following:

Definition 20.5. A fully polynomial approximation scheme (FPAS) is one where the runtime is polynomial in n and $\frac{1}{\varepsilon}$.

This still has its limits — as ε gets really tiny, the runtime becomes infeasible. But it at least gives hope you can run for some decent ε.

§20.4 Knapsack

Next we'll see a general technique for developing fully polynomial approximation schemes for a large class of problems. We'll go back to the knapsack problem.

Example 20.6 (Knapsack)

We have item sizes s_i and profits p_i, and some knapsack size k. Our goal is to find the largest profit that fits into the sack — in other words, we want to maximize $\sum_{i\in Y} p_i$ (by choosing a set Y of things to put in the knapsack), subject to the condition $\sum_{i\in Y} s_i \le k$.

Before, we talked about how there's no *additive* approximation for knapsack, because of scaling. But we're going to give something almost as good — we can't get an exact algorithm, but we'll get arbitrarily close.

§20.4.1 Dynamic programming for small profits

We'll use an idea we've seen before, of starting to concentrate on small integers. Suppose that we have small integer profits — so we know that the optimum solution is some small (polynomial-value) integer, made up of a sum of small integers, and that's all we need to find.

In this sort of setting, where we're working with small integer values, it's really natural to dust off dynamic programming. It's pretty clear that figuring out whether it's possible to achieve a profit of 1 is straightforward (for very small values, met by just a single item, you can just look and see). Larger profits can be assembled from smaller profits, and this satisfies the subproblem optimality structure that you want for dynamic programming.

Definition 20.7. We define $B(j,p)$ as the smallest set of items among items $1,\dots,j$ achieving profit at least p.

We claim that from this definition, you can tease out a pretty natural dynamic program. Suppose that we've already solved the relevant subproblems, and now we're looking for $B(j+1,p)$ — the optimal subset of items $1,\dots,j+1$. How can we construct this set? It'll either contain item $j+1$ or not. If it doesn't contain the $(j+1)$th item, then it's just a subset of the items $1,\dots,j$, so it has value $B(j,p)$. The other possibility is that we do include the $(j+1)$st value in this optimum subset. And if we are including that, then what's the rest of the solution going to be made of? Now we look at the smaller set of items $1,\dots,j$, but we know we're already getting p_{j+1} profit from the $(j+1)$st item, so we can subtract that off and say we only need a smaller profit from the previous items. But we've also used more space, so we need to add in the space we just took for that item — so in this case, we get $B(j,p-p_{j+1})+s_{j+1}$.

These are the only choices, and we take the better one; so

$$B(j+1,p)=\min\{B(j,p),B(j,p-p_{j+1})+s_{j+1}\}.$$

So this gives a dynamic program for small integer profits.

How much time does this take? We're going to have to run this program until we reach the optimum profit p_{OPT} that we're looking for. We'll start at $p = 0$, where it's easy. Then to get the next value of p (to increment p by 1), we need to work through n distinct first entries in the dynamic programming. So this will take overall time $n \cdot p_{\mathsf{OPT}}$.

How do we know when to stop? We stop when $B(n, p) > k$ — at that point, we know we can't achieve that profit.

> **Student Question.** *Doesn't this only check for getting the exact price?*
>
> **Answer.** We defined this as achieving profit *at least* p, not exactly p. The dynamic program handles this as long as we define $B(j, p) = 0$ when $p < 0$.

Have we proven $\mathsf{P} = \mathsf{NP}$? No (which is a shame, because David would love to retire with a Turing award). The problem is that this is only for *small* integer values. The input could contain numbers of exponential value in the input size. So in fact, what we have here is something we've seen before when talking about flow algorithms — we've given a *pseudopolynomial* algorithm, which is not a fully polynomial algorithm (it's not even a weakly polynomial algorithm). If we could get the runtime down to $n \cdot \log p_{\mathsf{OPT}}$, then we would have our Turing award, because that *would* be a polynomial-time algorithm.

§20.4.2 Handling large numbers

What about all the other cases, where the numbers are large (or not integers)? This is where we can't hope to solve the problem exactly, but we have something to base ourselves on. And we're going to use a technique we've seen before, namely *rounding* — the idea that you can ignore the less significant bits of your problem and still get pretty close to the optimum solution.

Let's suppose that $\mathsf{OPT} = p$. Imagine that we scale all the profits by replacing

$$p_i \mapsto \left\lfloor \frac{n}{\varepsilon p} \cdot p_i \right\rfloor.$$

What does this do to our optimum? Floors are confusing, so let's ignore that for now. If we just did the proportional scaling, then the value of the optimum would become $\frac{n}{\varepsilon}$ — if we scale all the profits by some amount, then we're also scaling the optimum by the same amount.

Now, how does this change when we take the floor? The number of items (which is n) — the profit is made of a sum of individual profits, and there's only n items; by taking the floor, we're taking off at most 1 from each item, so at most n in total. So the new optimum is at least $\frac{n}{\varepsilon} - n$.

So the new profit is basically $\frac{n}{\varepsilon}$. We lose an n, but if ε is small, then the amount we lose by taking the floors is only a small fraction of what the profit should be, so the floors shouldn't matter very much.

Now that means there exists a solution with this value $\frac{n}{\varepsilon} - n$. And we can find it using our pseudopolynomial algorithm, in time $n(\text{scaled } p) = n^2/\varepsilon$.

We found a solution of this value; it might not be the optimum (if it always were, we'd have $\mathsf{P} = \mathsf{NP}$). What's the *unscaled* value of that solution? The scaled value was $(n/\varepsilon - n)$. When we unscale, we don't get to un-floor (we don't necessarily get much back by removing the floor), but we do still multiply back by $\varepsilon p/n$ (the reverse of our scaling factor). And so now the ns cancel, and the ε in the first term cancel, and we get

$$\text{unscaled value} \geq \left(\frac{n}{\varepsilon} - n\right) \cdot \frac{\varepsilon p}{n} = (1 - \varepsilon)p.$$

(It's a $\geq$ because we might gain some stuff back from the removal of floors; but we certainly gain back this factor.)

How do we know what p to scale down by, if we don't know p in the first place? We'll instead use an *estimate* on p. What would the effect of that be on our approach? Suppose we use an estimate p'.

First, let's think separately about estimates that are too large and too small. Suppose that our estimate is too large. What will that do to this approach? If p is too big, then the error is going to increase beyond ε — the new optimum will be smaller than n/ε, so subtracting n will introduce too much error into the solution. So using an upper bound on p is not good.

However, if we use a *lower* bound on p (something that's smaller than p), then subtracting n is still fine — we get a larger scaled value, so subtracting n has *less* of an impact on the error, and we end up with a better than ε approximation. So we want a lower bound $p' < p$, so that we still get our ε error.

Where can we find a lower bound? We can just take the highest-profit item that fits in the knapsack. This is certainly a feasible solution, so it provides a lower bound on OPT.

What are the consequences of using a lower bound? If there were no penalty, we could just take $p_i = 1$. But what goes wrong if we take too small a lower bound is that the algorithm becomes slower — you're not scaling things down by as much. So if we take a small scaling factor, then we don't scale things down by very much, and the individual profits are still large, and the optimum scaled profit is still large. And the runtime of our algorithm depended on the optimum profit. So if we make the optimum profit really large, then we're going to have to run the algorithm for a long time.

How small is this compared to the correct p? It's at least $\frac{1}{n}$ of the optimum — the largest-profit item has to make up at least a $1/n$-fraction of the optimum profit. So $\max p_i \geq p/n$, which means our runtime gets worse (because now we're going to be possibly running up to np) by a factor of n — so it'll be n^3/ε.

So put another way, we now have to run up to not a potential profit of n/ε, but really to n^2/ε — we're looking at what OPT got scaled to, and it got scaled to something like n^2/ε, so we have to run all the way to profits of that size.

But still, this is a fully polynomial approximation scheme for knapsack.

§20.4.3 Improving the runtime

It is a bit unsatisfactory that it's slower; how might we speed it up? If p' turns out to be too small, maybe we can just make it bigger. The consequence of p' being too small is that the algorithm is going to run too long, because it keeps finding profits that are too big. So that gives us our signal — if we use a p' that's too small, then we keep finding profits bigger than n/ε, but as soon as we do we know p' is too small, so we should make it bigger.

There's not an obvious way to do that in the middle of the dynamic program, but you can throw away your work and start with a new p'.

So the idea is that we *start* with $p' = \max p_i$. But then we run the dynamic program until you find an impossibly large feasible solution, which tells you your p' is too small.

If we find p' which is less than p, but bigger than $\frac{p}{2}$, then we'd get our n^2/ε runtime — we just lose a factor of 2, because we're taking p to be half its intended value. So in other words, if we can find a $(1/2)$-approximation, then we can find any approximation fast. So this sort of reduces the problem of quickly finding an ε-approximation to quickly finding a $1/2$-approximation.

And how do we do that? It's the same challenge, but now we can fix $\varepsilon = 1/2$ and not worry about it anymore. What we're going to be worried about is that maybe p' is too small — $p' \leq p/2$ (that's going to make the optimum too large in our scaled solution, and we'll run too long).

But if that's the case, then our scaling takes the optimum solution to something in

$$\left[p \cdot \frac{n}{\varepsilon p'} - n, p \cdot \frac{\varepsilon n}{p'}\right] = \left[\frac{2np}{p'} - n, \frac{2np}{p'}\right].$$

And if $p' < p/2$, then even the small end of this interval is going to be at least

$$\frac{p \cdot 2n}{p/2} - n = 3n.$$

On the other hand, as soon as $p' > p$, we know OPT scales to something at most $n/\varepsilon = 2n$.

So we have a way of distinguishing these cases — if our guess is too small then we'll find an optimum at $3n$, and if it's too big, then we will not.

So now we can just apply repeated doubling — we start at $\max p_i$, and keep hunting for size-$3n$ solutions, doubling p' each time we find one. Our p' is too small if we find a solution of value $3n$, so we double it and try again. We repeat this until we get a solution that's *not* bigger than $3n$, and that tells us that our p' is big enough (bigger than $p/2$).

And the number of doubling steps we need to perform is at most $\log n$ — we started within $\frac{1}{n}$ of OPT, so $\log n$ iterations suffice. And since we're only targeting an objective value of $O(n)$, each of the testing steps only takes n^2 time, so this just adds an $n^2 \log n$ runtime to hone in on an appropriate value for our scaling factor, and then another n^2/ε to find the approximate solution.

So in total, we get $O(n^2 \log n + n^2/\varepsilon)$ runtime for a $(1+\varepsilon)$-approximation.

As a flag, this binary search is only approximate. We can't use this to find the *exact* value of p (if we could, we'd have P = NP). The point is that as we do this repeated doubling, we have this condition that at some point we'll be below $n/2$, and we'll at some point notice we overshot. But we don't know where in that doubling interval the actual value of p is.

§20.5 FPASs and pseudopolynomial algorithms

We demonstrated this rounding technique on a particular problem, but as a warmup, we also saw that we could solve the problem for small integer input values. This can be seen as a sort of weakness of the knapsack problem — it's NP-hard, but only if you put really huge numbers into the input instance. So in fact, what this integer solution showed is that knapsack is what's called *weakly* NP hard — it only becomes NP-hard when the numbers are large. The contrast is with strongly NP-hard problems, which are hard even if the numbers are small. Of course, one large class of such problems is ones that have no numbers (e.g. Hamiltonian paths or set cover) — there there's no dependence on the value of the numbers, because there are no numbers.

But what this demonstrates is that in order to use this technique, you need a pseudopolynomial algorithm to start with, and only weakly NP-hard problems can have them (if you're strongly NP-hard, then you're not polynomial even with small numbers, so there can't be a pseudopolynomial algorithm).

What we've shown is if you have a pseudopolynomial algorithm, you can use this rounding approach to get a fully polynomial approximation scheme. It turns out that pretty generally, the converse is true — if there is a fully polynomial approximation scheme, then you can get a pseudopolynomial algorithm. This is actually pretty easy to see — suppose that your instance has numbers bounded by t. If the objective is a sum of those numbers (which has been the case for every problem we've looked at so far), then it's bounded by nt, where n is the number of items (or numbers) in the problem. (We assume everything is an integer.)

So we can find a $(1+\varepsilon)$-approximation with $\varepsilon = \frac{1}{1+nt}$. Anything this close will in fact be an optimal solution.

But the runtime is going to be polynomial in the input size and nt, so it's going to be a pseudopolynomial algorithm.

So there's a very tight relationship between FPASs and these pseudopolynomial algorithms, which are fast when the objective is a small integer. In fact, it's been worked out from a complexity-theoretic perspective that this is the only way to make pseudopolynomial algorithms (with proper assumptions). When you do complexity theory, there's lots of bizarre problems that don't meet natural assumptions. But at least for a very broad class, there's this direct relationship between FPASs and pseudopolynomial algorithms.

§20.6 Enumeration

Question 20.8. What do we do about problems that don't have pseudopolynomial algorithms?

For this, we'll start introducing the technique of enumeration, which is a fancy term for brute force.

Example 20.9

We'll return to P $||$ $c_{\max}$ (our scheduling problem).

In our examples from last time that created large gaps (between our solution and the optimum), the bad jobs were the really big ones — the problem that we had with our greedy algorithm was that maybe you put lots of tiny jobs onto the machines, and then you throw on this one big job. Even in the analysis, the problem was that on top of the clear lower bound on OPT from the average, we added in the size of the last job that was scheduled; and maybe the last job all by itself is equal to the optimum, which is what gave the factor of 2 error.

An obvious way to fix that is to do something else for the large jobs. Imagine that we schedule the k largest jobs *optimally*, by brute force — so we take those k big jobs, and just try all possible solutions to putting those onto machines (one of those is the correct solution). And then we greedily schedule the rest of the jobs. Hopefully the rest of the jobs are small, so their p_js won't contribute so much to the value of our solution.

Claim 20.10 — If we do this, then our algorithm achieves $\mathcal{A}(I) \leq \mathsf{OPT} + p_{k+1}$.

This may seem completely obvious, but it isn't — we have to consider two cases. Let's consider our algorithm's last finishing job (the one that completes last), which has some completion time c_j. If it is one of the k largest jobs, what can we say? We built out this schedule, and in that schedule the last job to finish is one of the k largest jobs. Then that means the time to complete all the jobs is equal to the time to complete the k largest jobs. And we scheduled that time optimally, so we are OPT.

What if it's *not* one of the largest k jobs? In that case, we can say

$$\mathsf{OPT} \geq \frac{k}{m} p_{k+1},$$

because there are k jobs that are at least this big. And so the total work is at least kp_{k+1}, which means the average work is at least a $1/m$-fraction of that. (We've already seen the average load is a lower bound on OPT. And there are at least k jobs as big as p_{k+1}, so the total work of the largest k jobs is at least this.)

But that implies that $p_{k+1} \leq (m/k) \cdot \mathsf{OPT}$. And that means the job that finished last is at most $(m/k) \cdot \mathsf{OPT}$. If we go back to our analysis of the greedy algorithm, that means our completion time is

$$\mathcal{A}(I) \leq L + \frac{m}{k} \cdot \mathsf{OPT} \leq \left(1 + \frac{m}{k}\right) \cdot \mathsf{OPT}$$

(where L is the average load, as before).

So what kind of ratio is this? Well, it depends. What we've gotten here is a polynomial approximation scheme for constant m — if I have 10 machines and I want a particular ε-approximation for scheduling jobs on those 10 machines, then I set $k = 10/\varepsilon$, and I end up with an ε-approximation.

So this is at least a start — brute force on the larger jobs takes care of the worst offenders in our approximation approach, and lets us get a polynomial-time approximation schee when the number of machines is a constant. Next class, we'll show how we can do a cleverer kind of enumeration that gives a polynomial-time approximation scheme no matter how many machines there are. We use the same idea, that small jobs — jobs of size less than ε times the optimum — don't matter, in that you can schedule them greedily.

To make this point more carefully, what we're saying here is that if our last completing job is not one of the big jobs, then our error is determined by its size. So this tells us we can schedule small jobs greedily without fear of messing up our approximation. All we need to do is figure out how to schedule large jobs efficiently, and that's what we'll talk about on Friday.

§21 November 1, 2024

§21.1 Scheduling and enumeration

§21.1.1 Constant number of machines

We're looking at $P \parallel C_{\max}$. We saw a 2-approximation by working with two lower bounds — the average load and maximum job size. We did an analysis where we looked at the last job to finish, and supposed that it was added to a machine with load L, which took its cost to $C_{\max} = L + p_j$ (where p_j was the processing time of that job). But $L \leq \mathsf{OPT}$ (because L is at most the average load), and similarly $p_j \leq \mathsf{OPT}$. So we get $C_{\max} \leq 2\mathsf{OPT}$ (and we slightly refined this to $2 - \frac{1}{m}$).

Last time we started talking about *enumeration*, where we honed in on this second piece and said we may be able to arrange for that last job we add to be small — to be much smaller than OPT, in which case the total bound would do much better. (We'd still get OPT from the average load, but the amount by which we go above OPT is determined by this last job that we added.)

Last time, we had the idea of scheduling the largest k jobs according to the optimal schedule. We misspoke a bit — we said to schedule them optimally, but it's actually better to talk about choosing the scheduling of these k jobs that's optimal when considering all n (with the rest scheduled greedily). And then we assign the rest greedily. We then consider two separate cases, about what is the actual last job to finish that defines $C_{\max}$. If it's one of these jobs we scheduled according to the optimum schedule, then our solution is achieving exactly what the optimal schedule would be. So the only way we'll be off is if one of these other jobs we assigned greedily is the last job to finish, and it contributes p_j to the bound.

But because we scheduled the largest jobs separately, we know that $\mathsf{OPT} \geq \frac{k}{m} p_{k+1}$, because there are k jobs of size bigger than p_{k+1}, so this is a lower bound on the average load, and therefore a lower bound on OPT. This means $p_{k+1} \leq \frac{m}{k} \cdot \mathsf{OPT}$. In our naive analysis we just had $p_j \leq \mathsf{OPT}$, but now we have this scaling factor. That means we achieve a bound of

$$C_{\max} \leq \left(1 + \frac{m}{k}\right) \mathsf{OPT}.$$

As long as m is a constant, we can set k to be a bigger constant and make this number as small as we like.

What's the runtime for this algorithm going to be? Whatever we choose k to be, we have to brute force this first step. For each of these k jobs, we need to put them on one of the m machines, and we'll be trying all the possibilities. So here there are going to be m^k possibilities. And for each of those m^k possibilities, we have to do this greedy assignment of the rest, which is going to take us n time. And we need to set $k = m/\varepsilon$, so we get a runtime of $n \cdot m^{m/\varepsilon}$.

§21.1.2 Arbitrary numbers of machines

Today, we're going to figure out an approximation algorithm using enumeration, but for an *arbitrary* number of machines — so our algorithm will have runtime polynomial in m, not just n.

First, we'll assume that the optimum time $C^*_{\max}$ is known. We're going to try to create a schedule that finishes in time $C^*_{\max}$. We're moving away from an optimization problem to more of a decision problem (or a feasibility problem — we're trying to find a schedule that's feasible for finishing in this amount of time).

Have we made the problem any easier? No — if we can solve this problem, then we can also solve the optimization problem by binary search on $C_{\max}$. You might worry what level of precision we need, but since we're only working on approximation algorithms anyways, a little bit of loss in precision due to not quite finishing binary search will be swallowed by the error in our approximation algorithm.

§21.1.3 A special case — k distinct sizes of jobs

So how do we solve this? Let's start with an easier problem — let's suppose there are only k distinct sizes of jobs. (We'll see soon why this is easier.)

Then we'll be able to develop a dynamic program to optimally solve the problem. Notice first that an instance is just a count of the number of jobs of each size — we're basically condensing all the jobs of the same size into a count, instead of listing them separately. A machine schedule is the same — to describe what you put on a machine, you also only have to specify how many jobs of each size you put onto that machine. But of course there's an extra constraint — that

$$\sum(\text{count of size}) \cdot \text{size} \leq C^*_{\max}.$$

And an overall schedule is a collection of machine schedules.

Let's call a combination of jobs you can put on a machine as a machine *type*.

What's nice is we can count all these things. How many distinct instances are there? Well, I can have up to n jobs of each size, and k sizes; so there's at most n^k distinct instances. And by exactly the same reasoning, there are at most n^k distinct machine types.

How many distinct schedules can there be, then? That's too big a number. So let's not ask that question. Instead, let's use *these* numbers to think of a dynamic program for a feasible schedule. The dynamic program is going to answer the following question:

> **Question 21.1.** Which instances can be feasibly run by s machines, where $s = 1, \ldots, m$?

How do I figure out which instances can be feasibly run using a single machine? These are just the machine types (by definition).

Now suppose I know which instances can be feasibly run on a set of s machines; how do I figure out which can be run on a set of $s+1$? For every s-feasible instance, adding any one machine type gives an $(s+1)$-feasible instance.

How long does it take to do this — how much work do we need to do to go from s-feasible instances to $(s+1)$-feasible instances? There's at most n^k feasible instances for s machines, and at most n^k machine types; so there's $n^k \cdot n^k = n^{2k}$ combinations to try, and for each of them we have to do n work to update our counts. So there's n^{2k} pairs times n work, which is n^{2k+1}. And then you have to do this for $s = 1, \ldots, m$, so this gives you an overall runtime of $n^{2k+1}m$ (to enumerate all the instances that can feasibly be solved by m machines).

§21.2 Geometric rounding for job sizes

So we have a polynomial algorithm, as long as there are k distinct job sizes. The problem is that there aren't k distinct job sizes. But can we take an instance of our problem and get the number of distinct job sizes to be small, given that we're only looking for an approximation? We'll do some sort of rounding.

So far we did arithmetic rounding, where we divided everything by the same amount. That doesn't always mesh well with relative approximation algorithms — the problem is that if you have a small number and take its floor, you're changing its value by a lot. So we want a different way of rounding.

We're looking for a $1+\varepsilon$ relative approximation. This means two sizes within a $1+\varepsilon$ factor of each other are basically the same, for the purposes of such an approximation. This means we're going to apply a sort of *geometric* rounding.

So if two jobs differ in size by a *ratio* of at most $1+\varepsilon$, then they're indistinguishable for a $1+\varepsilon$ approximation (in the sense that we don't care which value you use). This is promising — it suggests we can stop worrying about jobs that are very close in value. Unfortunately, numbers have infinite precision. What if the sizes of our jobs are 1, $1/(1+\varepsilon)$, $1/(1+\varepsilon)^2$, $1/(1+\varepsilon)^3$, and so on? Then none of these would be close to any of the others according to this perspective, so it doesn't seem like geometric rounding by itself is going to simplify our problem.

But we've already introduced the trick that will overcome this problem. With these ever-increasing powers, the jobs are getting very small. And what can we do with small jobs? We've seen that we can sort of ignore them — if we schedule all the big jobs first, then the small jobs only introduce a small error. So we'll combine these ideas — that you only have to schedule the big jobs, with this geometric rounding idea so that we only have few sizes of big jobs. And then we'll take care of small jobs in the old greedy way.

§21.2.1 The algorithm

We'll say all jobs of size at least $\varepsilon C^*_{\max}$ are *large*, and the others are *small*. We're going to carefully schedule the large jobs and greedily schedule the small jobs, just as we did with the first enumeration algorithm.

First, we're going to schedule the large jobs carefully. For this, we round each upwards to the closest power of $(1+\varepsilon)$ (this means we're going to make every job bigger, but by at most a $1+\varepsilon$ factor).

If we've done this, how many different sizes of jobs do we have? Our large jobs start at size $\varepsilon C^*_{\max}$, and the biggest one is going to be at most $(1+\varepsilon)C^*_{\max}$. So the number of different sizes that could be in this range is roughly

$$\log_{1+\varepsilon} \frac{1}{\varepsilon},$$

since we're only looking at powers of $1+\varepsilon$ between two numbers whose ratio is $1/\varepsilon$. And we have

$$\log_{1+\varepsilon} \frac{1}{\varepsilon} = O\left(\frac{\log 1/\varepsilon}{\varepsilon}\right).$$

Now we can use our limited-sizes algorithm for these big jobs — we saw the runtime of that is n^{2k} where k is the number of distinct job sizes, so this has runtime $n^{\tilde{O}(1/\varepsilon^2)} \cdot m$.

And now we're home free — for any specific ε, this is a polynomial. So this gives an optimal solution to a rounded problem whose optimum is at most $1+\varepsilon$ times the original problem, because we just rounded up every number by at most $1+\varepsilon$.

We're not quite done — we still have the small jobs to take care of. But now we just schedule the small jobs greedily. And what we said about the schedule we just found is that either the last job to finish is large, which implies that we have a $(1+\varepsilon)$-approximation; or else the last job is small, which means our runtime is at most the average load (that's the L from the original analysis) plus the size of this one small job added to that average load, whihc is at most

$$C^*_{\max} + \varepsilon C^*_{\max} = (1+\varepsilon)C^*_{\max}.$$

So we're done. (We gave an algorithm that finds a feasible solution for a specified $C_{\max}$ if one exists; if you have this, then you can binary search for the optimum $C_{\max}$, within some small error.)

§21.3 General comments

What we've done here is combined the rounding idea (which we used for getting FPAS for knapsack, or other things with pseudopolynomial algorithms) with enumeration, which works even for problems that are not weakly NP hard. Last time we saw that a pseudopolynomial algorithm demonstrates your problem is only weakly NP-hard. Enumeration can work even for problems that are strongly hard. But the runtime you get, as we saw here, is polynomial in the input size but not ε. And if the problem is strongly NP-hard, that's the best you can hope for.

Last time, we talked about how a pseudopolynomial algorithm lines up with weakly hard problems, and rounding plus that gives you a FPAS; conversely you can turn a FPAS into a pseudopolynomial algorithm. So weakly NP-hard and FPAS kind of go together.

There is a pretty general complexity-theoretic result that says for a very broad class of problems, this enumeration technique is kind of the most general way to get a polynomial approximation scheme. It's phrased in quite abstract language in terms of what you're enumerating over and so on, but the idea is you always have to find some hard kernel of the problem, solve that by brute-force in non-polynomial time, and then you kind of fill in the rest without disrupting the solution very much.

§21.4 Relaxation and the travelling salesman problem

Now that we've seen some general techniques that get within $1 + \varepsilon$ of the optimum, we'll move back to problems where this is not possible — problems that are called Max-SNP hard or APX-hard, where it's been shown that if you can approximate them arbitrarily closely, then $\mathsf{P} = \mathsf{NP}$. So we'll look at problems where we have lower bounds on how well you can approximate them; nevertheless we want to get upper bounds as close to those as possible.

We'll finally see one technique that's very difficult for MIT students to internalize, which is called *relaxation*.

The idea behind relaxation, which we really should pay more attention to, is that if you're working on a problem and it's too hard, then you should just work on something easier.

We'll illustrate this via the travelling salesman problem.

Example 21.2 (Travelling salesman)

We've got a graph with edge weights, and we want to find a cycle that visits each vertex exactly once and returns to the start, minimizing the total edge cost.

That's the travelling salesman problem in full generality. It's also too hard. Forgetting about approximations, just finding a cycle that goes through all the vertices once and returns to the start is known as the Hamiltonian Cycle problem; and even that is NP-hard. So if you can't even find the structure, there's no point in trying to optimize its cost.

So we don't work on the general travelling salesman problem. Instead, we focus on special cases. One of these is what's known as metric TSP.

Example 21.3 (Metric travelling salesman)

In metric TSP, the graph is a complete graph (there's an edge between every pair of vertices), and the edge lengths are a metric, meaning they satisfy the triangle inequality.

This is actually equivalent to saying we're allowing multiple visits to a vertex, and the graph is undirected. A metric has to be symmetric, so the distance from a to b is the same as b to a; so we can represent that with an undirected graph. And if we allow multiple visits to a vertex, they're allowed to visit it as a waystation

between two vertices; this means the effective distance between two vertices is the shortest path in that graph. So if you allow revisiting vertices, then you have a metric; or we could just impose a metric from the beginning.

Metric TSP has seen a lot of interesting work, even in the past decade. Solving metric TSP is NP-hard, and actually max-SNP hard — there's some number that you can't do better than, and this is true even if the edge weights are just 0 and 1.

But we can do some very interesting approximations.

§21.4.1 General relaxation

Now we get to the key idea of relaxation. The idea of relaxation is that you have a hard problem. You find an easy problem that is 'similar,' and solve it. What do we mean by 'similar'? In general, what you do is relax the set of constraints — the constraints define some set of solutions that is feasible for the hard problem. You loosen some of the constraints to make the problem easy, and get a larger set of solutions that are feasible for the easy problem. Since the problem is easy, you can find a solution to the easy problem. And then you somehow (this is where the hard part is) round the easy problem solution to a hard problem solution.

So you start from your easy feasible solution, and transform it in some way that turns it into a solution to the hard problem. This transformation is where the approximation happens — you measure the approximation by comparing the rounded hard solution to the easy solution. Why does that work? It works because the easy problem had looser constraints, which means that the optimum of the hard problem is feasible for the easy problem. This means the optimum of the easy problem is *better* than the optimum of the hard problem. And so if your rounded solution is close to your easy solution, then it's also close to the optimum of the hard problem.

So that's the general principle. We want to take our hard problem and somehow loosen the constraints to get an easy problem. But knowing that we're going to be rounding the solution back to a solution to the hard problem, we don't want to throw away too much — we want to keep as much of the nature of the hard problem as we can, so that we can not change the optimum too much. So the question is always, what's the smallest change I can make to the problem to make it easy?

§21.4.2 Relaxation for Metric TSP

There's two versions of this problem — returning back to where you started or not — and when you talk about the TSP, people are kind of fuzzy. We originally defined the cyclic version, but let's talk about the path version for now.

What makes a path a path? Well, it has to connect up all the vertices. But is a path the only way to connect up all the vertices cheaply? If I just tell you that the important thing is to connect up all the vertices, a path is not what you would seek — instead you'd look at a *spanning tree*.

So the relaxation we'll look at is the minimum spanning tree. We'll take away the constraint of 'path' (you can think of a path as a spanning tree of maximum degree 2, so we've just relaxed the degree constraint on our spanning structure). We've now got an easy problem, and we can solve it.

But we're not done — now that we have a minimum spanning tree, we need to round it back to a path.

Another thing that's very important is to note that the minimum spanning tree cost is at most the TSP cost, because the minimum-cost path *is* a spanning tree — we've only loosened the constraints. All the old feasible solutions are still feasible, so we'll get one of those or something better.

So when we round, we'll be comparing our rounded solution to this minimum spanning tree, because this is a lower bound on the TSP — if we manage to stay close to the MST cost, then we're also close to the TSP cost.

How can we take a minimum spanning tree and find ourselves a path whose cost is not a whole lot more than the minimum spanning tree?

They did one of these experiments where they give lots of tools to solve a problem and it's hard, but if you take away these tools it becomes easy. Suppose I give you only the minimum spanning tree, and nothing else, and you want to use the MST to construct a path. We can just do this by an Euler tour, or DFS — we start at a vertex, go all the way around, and come back to the start. (Actually this is doing the cycle version.)

This will visit vertices multiple times, but that's okay because we're doing the metric TSP. General TSP is too hard — you can't even find a feasible solution, so there's no point in approximating — so we're just looking at metric TSP.

We want to eventually end up with a tour that visits every vertex once. So we'll start by doing a DFS, but after we find our DFS, we can shortcut across any vertices that we visit multiple times, because of the metric property — if hte DFS visits a vertex again, we just skip that vertex and go to the next vertex not visited, and we know the length of that path is at most that of the one from DFS.

And this traverses every edge twice (and then do shortcuts, but those only decrease our cost), so our cost is at most twice the minimum spanning tree, and therefore at most twice the TSP. So we have a 2-approximation to the (metric) travelling salesman problem.

So we found an easier problem that's similar enough, such that we could transform our solution without giving up a lot compared to the easy problem solution.

§21.4.3 The Christofides heuristic

We can do better than this — there is a better way to round a MST to a TSP. What we chose to do here was to take a DFS over the MST. Let's imagine we draw a bigger MST, so that we have more interesting cases to worry about. Here we did a DFS tour of this tree. But another way to describe what we did was that we doubled all the edges, which gave us a graph where every vertex had even degree; and then we took an Euler tour of that graph.

So a DFS is an Euler tour of the edge-doubled tree. (An Euler tour is a tour that visits all of the edges of the graph exactly once; there's a well-known theorem that you can find one if and only if all the degrees are even.)

The fact that you're visiting all the edges is what an Euler tour cares about. We don't really care about that, but if you're visiting all the edges, then you're certainly visiting all the vertices.

What Christofides said is, maybe there's a better way to make the tree Eulerian than doublig all the edges. So let's erase all the doubled edges, and go back to the starting problem. We've got this tree, and we'd like to take an Eulerian tour. But we can't take an Eulerian tour on a tree. Why not? The leaves all have degree 1, for one thing; and other vertices can also have odd degree. So if we want to make this graph Eulerian, what do we need to do? We need to make all the degrees even.

So we want to add edges to make the degrees even. We're working in a metric, so all edges are allowed — I can add whatever edges I want, they're part of my metric. And where do we need to add edges? We need to fix the degrees of the odd vertices. The even vertices are fine, I don't need to touch them. And to fix the odd vertices, I only need to add one edge to every vertex. Of course an edge has two endpoints; so what I need to do i sput edges on *pairs* of odd vertices. Ever odd vertex only needs to get one of those edges.

So what we need to do is find a *perfect matching* between odd vertices. As a small sanity check, can we do this? What if there's an odd number of odd vertices? No — this is the handshaking lemma, that the sum of degrees has to be even.

So there's an even number of odd vertices; we just need to pair them up and draw an edge between each pair, and this creates an Eulerian graph.

If we do it this way, what's the cost of the tour? We can use that Euler tour, and turn it into an actual TSP tour by shortcutting vertices; so the cost of the tour we construct is at most the cost of the MST plus the cost of the matching.

What does this motivate us to do? Well, it motivates us to find a *min-cost matching* between the odd vertices. Min-cost matching ins't a proble we studied in the class (we studied min-cost bipartite matching, using max-flow); it turns out min-cost non-bipartite matching is also tractable, and also uses an augmenting path algorithm, though the structures are a bit more complicated (you use these things called blossoms). But it is polynomial-time solvable.

But do we know there is a cheap min-cost matching in this tree? We want to show that the min-cost matching is cheap. We don't have a lot to leverage in that, but what we do have is the existence of the optimum TSP tour. Let's look at the optimum TSP tour. Now, some of the vertices on this tour are even, and some are odd. Suppose we have four odd vertices. We have a tour through the entire graph whose cost is OPT. WWe don't need the even vertices, so we can shortcut the TSP tour to only connect up the odd vertices. This only decreases the cost of the tour (because of the triangle inequality); so this tour on the odd vertices has cost at most TSP.

But it's a tour, and we want a matching. So how can we find that? We just take every other edge. In fact, the tour on the odd vertices is just the union of two matchings (alternating edges). That means one of those matchings costs at most half the tour cost, which is at most $\frac{1}{2}\mathsf{TSP}$.

So when we take the MST plus this matching, we're going to get at most $\frac{3}{2}\mathsf{TSP}$. This shows there are in fact constant-factor approximations besides 2 (all our algorithms so far have been 2-approximations, but you do sometimes get other numbers).

§21.4.4 Further work

This is not the end of the story. There's been a lot more work on the TSP problem, and recent work has taken us to constants less than 3/2 (i.e., $3/2 - \varepsilon$ for some $\varepsilon > 0$).

For a long time, we've known about the Held–Karp bound. This goes back to the idea that our relaxation is really relaxing the degree constraint — allowing our graph to have degrees greater than 2. The Held–Karp bound penalizes large degrees, trying to force your MST to have small degree. It can't quite force degree 2, but it pushes it away from star-shapes (which are very unlike cycles). This is *conjectured* to give a $\frac{4}{3}$ approximation, meaning that no one has found an instance where it does worse, but no one has been able to prove it always gets $\frac{4}{3}$; that is still open.

There's also work on the directed version, with the triangle inequality — this means $d(x, y)$ doesn't have to equal $d(y, x)$, but it still satisfies the triangle inequality. Equivalently, you can imagine it's a directed graph but you're allowed to revisit vertices.

In the directed version, for a long time the best-known was an $O(\log n)$ approxiation, which we're going to solve in the homework. Recently, people (here) pushed this down to $O(\log n / \log\log n)$, and we have no reason to believe it's not possible to do better (final projects are coming up, so there's lots of good open problems here).

Also, a really important breakthrough result in the 2000s — we talked about *general* metrics, where all you have is the triangle inequality. But if you're talking about a collection of points in the plane, this actually has a polynomial approximation scheme — you can get within $1 + \varepsilon$. This is the fact that you have geometry, and not just metrics — you divide the plane into lots of cells, from an approximation perspective what happens inside the cells doesn't matter, and you can dynamic program your way to find how the path moves through the different cells.

You can also do the same for planar graphs. A planar graph is one that you can draw on the board; the lengths may not be Euclidean, but you can use the same tricks of dividing the graph up into cells in order to create a dynamic program.

So there are lots of interesting cool algorithms, and lots of still open problems as well.

On the theme of relaxation, we'll finish early, and we'll do more intense relaxation on Monday.

§22 November 4, 2024

§22.1 Relaxation overview

Today is officially the most relaxing lecture of the semester; we are going to focus all our attention on relaxations. The idea of relaxation is that you loosen some constraints on your problem, which of course means that you only improve the optimum. Then you find an optimum of this easier (loosened) problem, which is *better* than the old optimum. That's straightforward; the tricky part is to then find some way to *round* the new optimum to a solution that is feasible for the original problem.

And the approximation ratio is better than

$$\frac{\text{rounded solution}}{\text{easier OPT}},$$

since we started with our hard problem, went to the easy problem (which only improved OPT), and then rounded; so the ratio between the rounded solution and the optimum of the harder problem is *better* than the ratio between the rounded solution and the optimum of the easier problem.

Last time, we saw a relaxation approximation for metric TSP. We used the minimum spanning tree relaxation. We showed a very naive factor-of-2 rounding, giving us a solution that was at most twice the minimum spanning tree (and therefore a 2-approximation). Then we saw Cristofedes's heuristic, which gave a $\frac{3}{2}$-approximation. Interestingly, this didn't exactly follow the above framework — we said we were paying the cost of the MST, plus at most half the optimum TSP (so we bounded the cost of the rounding based on the actual optimum, even though we couldn't get our hands on what it is).

We'll first talk about some problem-specific relaxations and roundings, and then talk about some general approaches.

§22.2 A scheduling problem — $1 \mid r_j \mid \sum c_j$

The next problem we'll look at is $1 \mid r_j \mid \sum c_j$. Here 1 means that we're just scheduling one one machine. We previously looked at $C_{\max}$, which is not interesting on just one machine. Here we're looking instead at $\sum c_j$, which is the sum of completion times (equivalently, the average completion time, up to scaling). And the thing in the middle is *release dates*, which says that you cannot process job j before time r_j. This doesn't seem like that big of an issue, but in fact it is a big issue — this makes the problem NP-hard.

So we can't find an optimum solution, but how well can we approximate? To develop some intuition, let's think about $1 \mid\mid \sum c_j$, without release dates (where anything can be processed at any time). If you're at MIT and just received 17 assignments for the week, which one should you do first to minimize your average completion time? You want to start with the shortest one — because the job you do first is adding to the completion time of *every* other job. So it's natural to use the smallest job for that.

So the heuristic this suggests is known as *shortest processing time first* (SPT); it's a very natural heuristic. (When we looked at $P \mid\mid C_{\max}$, we actually steered towards *longest* processing time first, because we were looking at a very different objective.)

Claim 22.1 — With no release dates, the shortest processing time heuristic is optimal.

Proof. Suppose the optimum schedule processes job j just before job k, but $p_j > p_k$. (If we are not scheduling the jobs in order of increasing processing time, that means somewhere, as we look at the jobs one at a time, we will find a job that is shorter than the one before it; that will be job k in this assumption.)

So we've got a bunch of jobs; then we've got this long job that takes time p_j, and then this short job that takes time p_k, and then we've got a bunch of other jobs.

We'd like to claim this is not an optimum schedule, and we can do that by exhibiting a better schedule — we just swap them. This is what's known as an *exchange argument* — if the schedule is not a SPT schedule, then we can improve it by swapping two jobs that are out of length order.

To be careful, we've focused on a local improvement — we've said that if you're locally not optimal, I can improve you. I'm not claiming that locally optimal implies globally optimal; I'm claiming that non-globally-ordered implies there is this local improvement. But that does prove the only optimal schedule has to be in increasing order of job length. □

§22.2.1 Relaxation by preemption

Now let's creep up on our original problem. What would get in the way of doing SPT in the original problem? You've got these release dates; maybe I want to run the shortest job first, but it's not released yet. So what if we run the shortest job that *has* been released?

This seems plausible, but now is where things get sticky. I'm running this shortest released job, but all of a sudden, a shorter job gets released. What should I do?

That's not the right question, because in $1 \mid r_j \mid \sum c_j$, if I'm running this job I have to finish it. But what do I *wish* I could do? Pause it and do the smaller one. Based on the SPT intuition, it seems plausible that if there's a shorter job available, maybe I should run that.

What does it mean if I'm pausing one job to run another? This is a framework we mentioned before, called *preemption*. So we can think about the preemptive version.

We claim that this is basically a *relaxation* of $1 \mid r_j \mid \sum c_j$ — if I allow you to interrupt jobs, then I am loosening the constraints on the problem, by permitting a new class of solutions that were not previously possible. I'm not requiring you to preempt jobs, so any schedule for the original problem is feasible even with the preemption added. So this is indeed a relaxation of the original problem.

But this problem is tractable, using exactly the algorithm we just described.

> **Claim 22.2 —** For this relaxation, shortest *remaining* processing time first (SRPT) is optimal.

Proof. We can prove this in pretty much the same way as before. Now we consider an optimal preemptive schedule that is not SRPT. What would that mean? That means there's a moment when we're processing job j, even though job k is available and has less remaining processing time.

Let's think about what that means. We've got job j, which we're processing for a little while; we may have processed it for a while before (because we allow preemption), and it also may get processed more later. But there's a moment where we're processing job j, even though k has less time left.

And we can also look at job k, which also may have gotten processed at various times; all that we know is that it has less total processing time (remaining) than job j. (Of course, there are other jobs being run at various different times, but we'll ignore those, and just look at the time spent on these two jobs.)

How can we improve on this schedule? One thing is that if we have two jobs that are both available to process, it's really silly for us to alternate between them — alternating between the jobs just pushes up *both* of their completion times. (One is still going to finish at the end of their combination, so all I can really control when the other one finishes. And if I interweave the processing, I'm making both of them finish

later, instead of just one.) So I should be greedy and finish off one of the jobs before I start the other (from this point). And then the question is, which should I finish off first?

I'm going to have to pay an amount corresponding to the sum of the work (for the second job). But the other amount I have to pay is determined by when I finish the first of these jobs, and that can be optimized by taking the smaller of these jobs to finish first.

So we should swap (from this point onwards) to completely finish k before starting j.

What this exchange argument proves is that if you are not doing SRPT, then you are not optimal; and therefore the optimal schedule is SRPT. □

§22.2.2 Rounding

So now we have a solution to this relaxation of the problem we actually wanted to solve. What we have left to do is rounding — we want to convert the preemptive schedule that we have here into a non-preemptive schedule.

The idea of rounding in general is to hope that there's enough connection between the relaxed problem and the original problem that the solution to the relaxed problem gives you some guidance about a good way to solve the original problem. What kind of guidance might we take from this preemptive schedule?

The idea is that we're going to run the jobs in the same order as in the preemptive schedule. It's not clear why this is a good idea — we may get into weird situations where I'm running the first job, but in the preemptive schedule I might stop and move on, and here I can't do that; so I might be preventing all sorts of other jobs from running.

The main idea is to look at the *finishing* times instead of the starting times.

So we take the preemptive OPT. Then if job j completes at time c_j, we insert an additional p_j spare time at c_j. So we take our timeline, where we're running various jobs (maybe there's job 1, and then I run job 2 for a while, and then 3, and then run 1 for a while until completion, and then run job 6). Then I cut my timeline at c_1, and insert p_1 time here.

Why? We're then going to run job j in that inserted p_j time (in our rounded solution). So in a sense, we're ignoring all the original time — we're inserting this magic new time, and only running jobs in the time that we're inserting.

Why does this work? Well, notice that we're *allowed* to run job j in the magic time that we inserted — it's after c_j, and certainly $c_j \geq r_j$, so running the job at time c_j is legal.

Now comes the key question — how much does the extra time delay some other job k? This takes the most insight. The key observation is that if the extra time for j delays the completion of k, that implies that $c_j < c_k$ (because you only get delayed by the extra time that was placed before you — extra time placed after you doesn't delay you). But that means that in the preemptive schedule, we ran all of job j before we finished job k (i.e., before time c_k). That used up p_j of the original timeline before c_k. And this means $c_k \geq \sum p_j$ where the sum is over all jobs j that finish before k (this is in the preemptive schedule).

So if a lot of jobs come before job k, then c_k is going to be quite large. And the *inserted* time before job k is exactly $\sum p_j$ over all such jobs j, by the rule for how we inserted extra time.

So what this means is that before job k, we're inserting at most c_k additional processing time, which proves that c_k at worst doubles. And since that's true for each c_k, that shows we have a 2-approximation.

> **Student Question.** *Why is this considered rounding?*
>
> **Answer.** David calls it rounding because we're taking a more relaxed solution and forcing it to be more constrained. When we round things to integers, we're taking a very relaxed integer, which has some stuff after the decimal point, and forcing it to not have stuff after the decimal point. So David thinks

it's metaphorically a good word for what's going on. Some examples of rounding will actually be very close to literally rounding, but we're not there yet.

§22.2.3 Improvements

As usual, the next question is, can we do better? The answer is yes. It turns out if we use some of the techniques from last time — rounding and enumeration — you can actually get a polynomial-time approximation scheme. A funny story is that in around 1998, David was with a friend at a conference and said, let's write a paper! And he asked, on what? So they pickd a scheduling problem, and went on a walk and solved it using rounding and enumeration. They submitted a paper, and at the same conference, 3 other groups submitted a paper that solved the exact same problem in the exact same way. These techniques had been around for 15 years, but no one had solved it; all of a sudden 4 groups did (resulting in a 13-author paper, because they were combined).

That's our scheduling theory story. An interesting observation here is that what this approximation algorithm does is it actually introduces unnecessary idle time into the schedule. You could have a job that's available to run, but you could choose to run no jobs in order to keep your machine free, so that when a smaller job is released later, the machine is free to run it. It's a bit counterintuitive you might want to ever leave your machine idle, but it turns out that's necessary to get a good schedule.

Student Question. *Could you do this online?*

Answer. There is certainly an online version of this problem. But you would probably have to constrain the class of instances. Otherwise, the adversary can sort of outwait you — we're going to study online algorithms in a couple of lectures, and then this will make more sense. But David is not certain and needs to think about it offline.

§22.3 ILP relaxation — vertex cover

For our next approach to relaxation, we'll look at a very general and powerful technique for doing relaxation, by example. We'll go back to vertex cover. And we'll solve this using linear programming! We've talked about how amazing a sledgehammer linear programming is; so let's use it to solve vertex-cover.

We have a variable x_i indicating whether we take vertex i or not; so we want to minimize $\sum x_i$. We have a constraint that we need to take at least one vertex for every edge, so $x_i + x_j \geq 1$ for all $(i,j) \in E$. And these are indicator variables, so $x_i \in \{0,1\}$ for all i.

So we're done, right? No — why not? What we've done here is written a constraint $x_i \in \{0,1\}$ that you cannot actually express in a linear program.

What if we instead wrote it as a constraint that you *can* express in a linear program — $0 \leq x_i \leq 1$ — but also sprinkled in $x_i \in \mathbb{Z}$? How's that for a linear program? Well, none of our analysis works if x_i is constrained to integers. This is what's known as an *integer linear program*. And by giving you an integer linear program that solves vertex-cover, I've demonstrated that integer linear programming is NP-hard.

So it's useless. But not really! One reason is people have developed really powerful ILP solvers, used all the time in practice. So if you can formulate your problem as an integer linear program, you can maybe throw it as a solver and get a fast answer.

But that's not interesting from a theoretical perspective, since those are worst-case exponential. But there's still something really interesting.

We're talking about relaxations; how might we relax an integer linear program? We can just take away the integrality constraints. So we're going to talk about LP relaxations of ILPs.

This is one nice thing about ILPs — if you express your problem as a LP, the relaxation is completely obvious (take away the integrality constraint). What's not obvious, though, is after you solve it, how do you *round* this solution?

This depends on the problem — we'll see a general mechanism, but in that case the analysis depends on the problem. And in many cases, even *how* you round depends on the problem.

In this instance, how might we round? One possibility is to round by rounding, i.e., to round to the nearest integer. What if we just take each x_i and round it to the nearest integer, which will be 0 or 1?

We claim this actually works — why? For any edge (i, j), we have $x_i + x_j \geq 1$, so one of x_i and x_j has to be at least $\frac{1}{2}$, and therefore rounds to 1. This means we get a vertex cover — for every edge, one of the endpoints is going to be taken in the vertex cover.

What about the approximation ratio? Well, we're going to use this trick of comparing our rounded solution to the value of the relaxed optimum. Here, the relaxed optimum gets a value of $\sum x_i$ (where the x_i's are fractional). How does our rounded solution compare to this? Each x_i at most doubles — if $x_i < \frac{1}{2}$ we round to 0, so it shrinks; only if $x_i \geq \frac{1}{2}$ do we round to 1, and that means we do at most a multiplication by 2.

So that means the sum of the rounded x_i's, which we denote by $\widehat{x_i}$, is

$$\sum \widehat{x_i} \leq 2 \sum x_i.$$

And because we had a relaxation — removing the integrality constraints makes for more feasible solutions, so the optimum gets better — the original vertex cover is feasible for the relaxed problem, which means the objective is better than the best vertex cover. So we get

$$\sum \widehat{x_i} \leq 2 \sum x_i \leq 2\mathsf{VC},$$

which means we yet again have a 2-approximation.

This whole technique even works for solving the *weighted* version of vertex cover, where it costs different amounts to buy different vertices. With the super-greedy algorithm it's not obvious what to do. But here it is obvious what to do — you just change the objective function to $\sum w_i x_i$. And exactly the same rounding procedure works, for the same reason as before.

§22.3.1 Improvements

As usual, the next question is, can you do better? There's been a long line of work trying to show that you *can't* do better.

Hastad showed that you can't beat $\frac{7}{6}$ (in 2001). Then it got improved to $10\sqrt{5} - 21 \approx 1.36$ by Dinur–Safra (who were some of the people who developed all the machinery of hardness of approximation and PCP and so on). Then it got improved to $\sqrt{2}$ by Khot–Minzer–Safra.

When complexity theorists can't prove something they want, sometimes you reduce it to something else you can't prove. Khot proved that you can't beat 2 if the so-called *unique games conjecture* is true. The unique games conjecture — David has looked at its definition and there's no game there. It's just another weird combinatorial problem that's called a game. But it's a particular problem that he invented and reduced vertex cover to, which spawned a whole industry of papers about the unique games conjecture. This is now a heavily studied problem in computer science. It has other ramifications as well, but one implication would be this lower bound of 2.

The bound of 2 only refers to the constant, but doesn't prevent you from having lower order terms. People have improved this 2-approximation to $2 - \Omega(1/\sqrt{\log n})$. You might laugh and think this is basically 2, but it turns out this matters a lot — this powers the best-known approximation algorithms for graph coloring. We got a lot of progress in graph coloring by using a slightly better than 2-approximation for vertex cover

as a subroutine. So there's active work on both of these — to strengthen the proof of hardness, and to do a better job on this lower-order term.

As one other interesting thing, the complement of vertex cover is — a vertex cover is a collection of vertices such that every edge has at least one end in the vertex cover. So the complement — all the other vertices — is an independent set. If we invert the definition, we've got a collection of vertices, and no edge has both its endpoints in this collection of vertices. So the complement is an independent set (a set with no edges between any two vertices).

This is another very classically studied problem (if you take the edge-complement you get Clique, which is the same problem and also very interesting).

But what's really fascinating about independent set is that with no qualifications, it's known to be NP-hard to get within a multiplicative $n^{1-\varepsilon}$ of the correct answer, for any ε. In other words, even if the independent set is a third of your graph, you can't find more than a tiny, tiny independent set (unless $\mathsf{P} = \mathsf{NP}$). In fact, this was the first problem to be shown really, really hard to approximate, using the PCP theorem (which you might see in a complexity theory class). Lots of other problems have been demonstrated hard by reduction from this one.

The reason to bring this up is to make the point that you can have two closely related problems (VC and IS) where if you could solve one optimally, you could also solve the other; but approximating one can be *very* different from approximating the other.

This may be counterintuitive at first, but a good explanation is that if you scale the solutions for two complementary problems, if one is really close to 0, then the complement will be really close to 1. In order to get a solution which is close to this optimum of 1, I can find a solution of value $1-\varepsilon$, and that'll be a very close approximation ratio. But at the other end if the solution is 0, the complement of this approximation will have value ε, which is of course infinitely far away from 0. So even though I got a great approximation for the latter, I got a terrible approximation for the former. This is what's going on with vertex cover and independent set.

Next we'll demonstrate some more powerful ILP relaxations.

§22.4 Logistics

David's goal is to have psets end around Thanksgiving. If we stay on track schedule-wise, then the last will be due before Thanksgiving; if we slip, one will be due after.

After Thanksgiving, the lectures essentially become optional. David will continue to give lectures on what he thinks are fun topics, on Mondays and Wednesdays but not Fridays. He will stop once it seems people aren't interested in listening anymore, which he'll judge by the number of people who show up.

The reason is so we have time to work on our projects. David will talk more about the projects soon. He tries not to get people thinking too soon because in general people pick their project from a topic we've already covered, and he wants us to expose us to more topics first. But the pset due before Thanksgiving will require us to submit a paragraph as a project idea, which they'll evaluate. So later in November is when we should start thinking about the project, and at that time David will give us some guidance.

§22.5 Facility location

In vertex cover, we had a simple linear program and the obvious relaxation, and a pretty straightforward rounding. Now we'll see a case where you have to work harder to get your rounding.

Imagine you're looking at openining up lots of marijuana distribution centers, and you want to figure out where to put them. Opening up a facility costs money, so you don't want to open up too many facilities.

But you do want to open facilities to serve customers. And if a customer has to go really far to make their purchase, that's a cost. So you want to balance these two costs — opening facilities, and allowing people to easily get to a facility, which is really about the cost of connecting each user to a facility (of course, you connect them to the closest one).

Example 22.3 (Facility location)

We have facility opening costs f_i, as well as assignment costs c_{ij}, and our goal is to open a set of facilities (at costs f_i) and then assign each customer to an open facility (at cost c_{ij}); and we want to minimize the total cost.

This is very broadly applicable — it also applies to where you want to put your network access points (you now have to run wires for all the computers that want to be connected into those points), and so on.

The general problem has set-cover as a special case. And set-cover is hard to approximate, so this is also hard to approximate.

So we'll focus on the *metric* case, where there is a metric involving the facilities and customers, so we can use the triangle inequality. That will make the problem easier to approximate.

§22.5.1 An integer linear program

We'll approximate it by using the ILP technique, so our first step is to develop an integer linear program that represents this problem.

We need some variables; y_i will be an indicator for opening facility i, and x_{ij} will be an indicator for assigning client j to (open) facility i.

Now that we have these variables, we can write down the ILP. The objective function is to minimize

$$\sum f_i y_i + \sum c_{ij} x_{ij}.$$

What are the constraints? This is an integer program, so $y_i, x_{ij} \in \{0, 1\}$. This allows us to use inequalities in weird ways, that wouldn't make sense for continuous variables.

For one thing, we have to assign every client to a facility — that's a requirement — so $\sum_i x_{ij} \geq 1$. Because we're in the $\{0, 1\}$-case, this is actually saying that one of the x_{ij}s has to be 1 (client j has to be assigned somewhere).

We also need $x_{ij} \leq y_i$, which is articulating that you can't go to a facility that isn't open — this is saying that in order for x_{ij} to be 1, y_i has to be 1. In other words, in order to assign j to facility i, facility i has to be open.

So this is our ILP. You can see how this decomposes the challenge of working with a new approximation problem — first you just have to figure out how to represent it. And once you've represented it this way, now we can relax; relaxation just means that we take $0 \leq x_{ij}, y_i \leq 1$ (instead of requiring them to be equal to 0 or 1).

§22.5.2 Ideas for rounding

All the work comes in the rounding. How do we round a fractional solution to the facility location problem?

As some intuition, what might you want to try, and what might go wrong? For one thing, we definitely can't round to the nearest integer — the reason it worked in vertex cover was that our constraints only involved two variables. Here we have a gigantic sum — you could achieve $\sum x_{ij} = 1$ by having each of them be $\frac{1}{n}$. So all could be tiny numbers.

There's one technique that's led to very good rounding algorithms (such as network design): identify a single fractional variable which can be rounded to an integer at low cost, and then rerun the linear program with that variable fixed at the value we found. That is a good technique to think about, but it's not one we'll use for this problem; here we'll round everything at once.

One attempt is to round up things that have value $\frac{1}{n}$; but then we're multiplying the value of things by n, which gives a n-approximation; that's quite poor.

What if we choose the x_{ij} that's biggest? We might have the same concern, where even though it's the biggest, it might still be really small.

This is where the convexity of linear programming becomes really frustrating — you want an integer, but really all your things are being smeared into tiny numbers.

To round, we're going to take two steps. We'll separately do some partial progress by getting rid of outliers, and then we'll do a separate step to finish.

§22.5.3 Filtering

The particular problem we'll worry about here is that we want to *fully* assign j to some *partial* assignment x_{ij} — we want to round up one of the x_{ij}s to 1. Now, what we're going to pay if we do that is a cost of c_{ij}. The cost that our fractional solution is paying can be seen as $c_j = \sum x_{ij}c_{ij}$, a weighted average of the c_{ij}s (we know $\sum x_{ij} = 1$, so this is a weighted average of the c_{ij} costs). Note that these c_{ij}s are in the objective function. So if we could actually assign at a cost equal to the average, we'd be doing well.

What could go wrong if we assign to something that has a value less than 1? We've got this average value, but maybe there are some very large c_{ij}s in that average, which are being scaled down to very small numbers by small x_{ij}s. So even if we have a really good average value, some candidate assignments might be very expensive. And we don't want those. So we're going to exclude them.

Prairie Home Companion used to have the slogan 'the place where all the men are strong, all the women are good-looking, and all the children are above average.' We know this can't happen, we can't have everything above the average. And moreover, not many things can be *far* above the average.

> **Claim 22.4 —** At most a $1/\rho$-fraction of the weight x_{ij} can be on $c_{ij} > \rho c_j$.

For example, only half the weight can be on things that cost twice the average. Why? Otherwise, just multiplying that half the weight by things that cost twice the average would give you more than the average already. (This is just Markov's inequality.)

So we'll just throw those things away — we'll set them to 0. Now that we've done that, any x_{ij} that remains costs at most ρc_j. So if we end up assigning j to i, we're paying at most ρc_j, and so we've got a good approximation to the assignment cost.

Unfortunately, we've just made the LP infeasible — it is no longer the case that $\sum x_{ij} \geq 1$. We're going to fix that by scaling up all the x_{ij}s by $1-1/\rho$ to compensate. We threw away a $1/\rho$-fraction, so if we multiply all of them by $1 - 1/\rho$, we still have $\sum x_{ij} \geq 1$.

But unfortunately, this is still not feasible, because the x_{ij} have been scaled up but the y_i's haven't. So I also have to scale up the y_is by $1 - 1/\rho$ in order to maintain $y_i \geq x_{ij}$.

Now I can assign each j to *any* open i with $x_{ij} \geq 0$, and pay at most $\rho \sum c_j$. That implies the *total* assignment cost is going to be at most $\rho \sum c_j$, which is the second term in the objective function (i.e., $\sum c_{ij}x_{ij}$).

So by filtering, I've made sure that the assignment cost is not a lot bigger than the fractional assignment cost.

§22.5.4 Facility opening

But that's sort of still looking ahead. I've ensured the assignment cost is going to be small, but I still haven't told you which facilities I'm going to open. So let's talk about that. (This is where we're going to use the metric property.)

To assign, I have to open facilities. The problem is that all the y_i might be small — which is the same problem as before. If all the y_i are $1/n$, then rounding one of them up might substantially increase our facility opening cost compared to the fractional solution.

So what we'll do is find a 'cluster' of nearby facilities whose total y_i sums to at least 1 — we're going to take a bunch of facilities together that add up to 1 open facility. But we'll make sure they're all close to each other.

Now we have this collection of one fractional facility, and we need to turn them into an integer. Of the facilities in the cluster, which one does it make sense to open? We'll just use the cheapest facility.

What's the cost of that? The cost is

$$f_{\text{cheap}} = f_{\text{cheap}} \cdot 1 \leq f_{\text{cheap}} \sum_{\text{cluster}} y_i \leq \sum_{\text{cluster}} y_i f_{\text{cheap}} \leq \sum_{\text{cluster}} y_i f_i,$$

which is in the objective function (the first piece). So if we're able to break the facilities into clusters which each add up to one unit, then we can afford to open the cheapest facility in each cluster.

(We'll finish next time.)

§23 November 6, 2024

§23.1 Facility location

§23.1.1 Review

Last time, we were in the middle of the facility location problem. We want to open some facilities and assign each customer to a facility, to minimize the sum of costs of the facilities you open and cost of connecting clients.

We wrote this down as an integer linear program, and then relaxed it. And we started figuring out how to transform a solution to the relaxed problem — which has a better objective value than the optimum — into an integral solution to the original problem. The first thing we did was separate out the x_{ij}'s for a particular client j and define $c_j = \sum x_{ij} c_{ij}$, the fractional connection cost of j (the average connection cost, weighted by the amounts by which that client is fractionally connected to those facilities). We argued that at most a $1/\rho$ fraction (by these weights x_{ij}) can be greater than ρc_{ij} — because if a lot of the connection cost is more than ρ times the average, that by itself would contribute too much to the average, and the average would be too large. So there are only a small amount of outlying facilities that are too far away, and we zero out their x_{ij}'s.

This breaks feasibility because we've reduced $\sum x_{ij}$; but we've reduced it to at most $1 - 1/\rho$, so we can recover feasibility by scaling up the remaining x_{ij} by a $\rho/(\rho - 1)$ factor. And that's where we left off last time.

After this filtering step, we want to think about actually opening up some facilities. Thanks to this filtering step, any facility that remains available to a client is not too far away; so it kind of doesn't seem to matter that much which facility we connect to — because the fractional objective function already assumes we're going to be paying this average cost c_j. So if we end up paying ρc_j, then we're not spending overall much more than the relaxed objective function.

§23.1.2 Facility opening

Now we started talking about facility opening. We want to again look at the cost of the facilities we're opening, and relate it to the cost in the fractional solution,which is $\sum f_i y_i$. If we pay something close to $\sum f_i y_i$, then we know we're close to the optimum. The problem is that all the y_i might be very small numbers; and if we turn any of them into 1, we'll be spending a lot more than that $f_i y_i$, because we're spending the full f_i instead.

In order to address this, the idea was to find a *cluster* of facilities with $\sum y_i \geq 1$. Then we can open the cheapest facility in that cluster, and pay at most $\sum f_i y_i$ (where the sum is over that cluster) — because $\sum y_i$ is 1, so this is just a weighted average of the f_i's in the cluster, so if we open the cheapest facility in the cluster, that will cost at most the weighted average.

So if we can do a good job of finding a cluster of facilities to open, then we can afford to open one facility in that cluster. Now, what cluster are we going to create? Well, if you think about it, a problem with the clustering is that we found this collection of facilities, which is our cluster; so we're going to open something in this cluster. But we also have all these different clients, who want to attach to different facilities in the cluster. And if we only open one of the facilities, all of a sudden it seems much more expensive to attach the clients whose facilities didn't open. So that drives us to be more careful in building our cluster.

This is where we'll really take advantage of the metric restriction. To build the cluster, we choose the *client* with the cheapest c_j (the cheapest fractional connection cost). And our cluster is going to be all of his fractionally connected facilities — in other words, $\{i \mid x_{ij} > 0\}$. Remember we threw away some connections that were too expensive; so what remains is a collection of facilities that are all pretty close to j, and we make a cluster out of all of them.

Why does this meet the requirement $\sum y_i \geq 1$? It's because of the constraint $\sum x_{ij} \geq 1$, saying there has to be a unit of connection coming out of j; and that gets absorbed by facilities.

So now we open the cheapest facility in that cluster, and we're going to zero all the others — the other facilities had some fractional weight, but we used it up to pay for the facility we've opened, so we can't reuse them in another cluster.

Now, what could go wrong if we zero out a bunch of facilities (making them not openable)? This means you're going to force some x_{ij}'s to become 0 — you're closing some facilities that other clients are relying on. So that seems to be a problem — what do we do with those other clients that are relying on the facilities you zeroed out? Well, we're just going to assign *every* such client (i.e., every client j with any $x_{ij} > 0$ for i in the cluster) to the just-opened facility. This is overkill, but it analyzes well. We're closing some sites in this cluster; any client that depends on any of the sites in this cluster, we assign *all* of them to the one we chose to open.

Now we're going to prove that this is relatively cheap. Here's the situation: in the center, we have the facility with minimum f_i; and we opened this cluster because we had some client j with the minimum c_j. And the cluster that we used was the set of facilities that j wanted to connect to.

Now, what do we know about all these facilities j was considering connecting to? We know the connection cost to j is at most ρc_j, because of the filtering step.

Now let's ask about how much it might cost us to connect some other j' — j' was planning to connect to some facility in the cluster, but we closed it, and we instead forced j' to connect to the facility that we opened. What is the cost of that connection? Well, this is where we can use the triangle inequality. Since j' was considering this site, we know the cost of connecting j' to that site is at most $\rho c_{j'}$ (by the same filtering rule). And we have a path of length 3 between j' and the facility that we opened (j' to that facility to j to the facility we opened). The first edge costs at most $\rho c'_j$. The edge from that facility to j costs at most ρc_j, again by the filtering rule. And finally, the connection from j to the cheapest f_i also costs at most ρc_j.

So we have one edge of cost $\rho c_{j'}$, and two of cost ρc_j; and we have the triangle inequality. So the cost of connecting j' to this facility is at most the sum of these three edge lengths.

So if j' wants a site i' in the cluster, then the cost of connecting it to i instead is at most

$$\text{cost}(j', i') + \text{cost}(i', j) + \text{cost}(j, i) \leq \rho c_{j'} + 2\rho c_j.$$

And to build the cluster, we chose the site whose c_j was minimum. That lets us simplify this — it means altogether, this is at most $3\rho c_{j'}$, because $c_{j'} \geq c_j$.

So through this clever construction, we've ensured our connection cost for client j' is only 3ρ times its average connection cost. So we're again relying on relating to the relaxed linear program — in the relaxed linear program, we have $c_{j'}$ cost for connecting j'. And we're only spending 3ρ times that cost. So locally, we have a 3ρ-approximation. (We have to look at how the different pieces affect each other, but from the perspective of j', we're only spending 3ρ times their connection cost.)

And we repeat this process — we open one cluster, which takes care of some facilities and clients, and then we open another, and so on, until there are no facilities left.

§23.1.3 The approximation ratio

Now we're basically done; we just need to finish assembling the different approximation ratios. During our filtering step, we multiplied all the y_i's by $1/(1-1/\rho) = \rho/(\rho-1)$. But then after doing that, we paid at most $\sum f_i y_i$ for opening everything. (We open up different clusters of things, but each facility is in only one cluster we consider.) So in total, we spend at most

$$\sum f_i y_i \cdot \frac{\rho}{\rho - 1}$$

for opening the clusters. Meanwhile, for connecting, each client j is connected at cost at most $3\rho c_j$.

So if we balance these, we want to set

$$\frac{\rho}{\rho - 1} = 3\rho,$$

which means we want $\rho = \frac{4}{3}$. Then both of these quantities come out to be 4 — so the facility opening costs and connection costs are both at most 4 times the corresponding part of the fractional objective. And therefore we have a 4-approximation.

If we think about it, we actually have a kind of bicriterion algorithm — by varying your choice of ρ, you can spend a larger multiple of the facility cost and smaller multiple of the connection cost, or vice versa. So if the connection cost is very large and facility cost very small, then you might want to move ρ to pay more attention to the facility cost and less to the connection cost, and get something better. But in all cases, balancing gives a 4-approximation.

The main point is that hopefully we see how the fractional solution gives a guide for how you can build an integral solution — where we're trying to carefully trace the fractional solution as we construct the integral one. And the value of the fractional objective function provides a bound against which to compare the value of the integral solution — when we claim a 4-approximation, we're arguing we're within 4 of the value of the fractional solution (we know OPT is between the fractional solution and the value we find, so if we're within 4 of the fractional solution, we're also within 4 of OPT).

Student Question. *Is there a general way to do something like this for ILPs?*

Answer. For many ILPs, the rounding technique is designed specially for the structure of that particular program — you don't get something for free. You still have to invest some creative thought. ILP gives you a *starting point* — you don't have to explore the entire space of approximation algorithms, just come up with a decent ILP, and then imagine starting from a fractional solution and rounding it (which is a much smaller search space to come up with algorithms from).

§23.1.4 Can we do better?

There's been a lot of additional research. Last time David checked, the state of the art was a 1.488-approximation. There's also a very interesting diversity of algorithms — some based on rounding (this one is probably based on a smarter rounding procedure), some based on local search, and all sorts of other things.

On the other side, there is a 1.463 hardness result — you cannot do better unless P = NP. This was one of the giant leaps forward when David was in graduate school — until then no one had any idea how to show it was difficult to approximate any problem. Now we have almost as rich a theory of hardness of approximation as we do of NP-hardness — for many problems we have strong lower bounds on how well you can approximate.

§23.2 Max-SAT and randomized rounding

Now we'll come back to the question for whether there's a general technique for rounding ILPs. There is one, and we'll apply it to another famous problem.

Example 23.1 (Max-SAT)

Suppose we have a collection of OR clauses over a bunch of Boolean variables. Satisfy as many clauses as possible with one assignment.

This really should be called Max-CNF-SAT. In SAT, we have a collection of clauses, and want to satisfy their AND — to find an assignment of variables such that all the clauses are true at the same time. That's just a decision problem — can you satisfy all the clauses or not? Max-SAT instead asks to satisfy as many clauses as possible.

Of course this problem is NP-hard, because if you could solve it, you could solve SAT (check whether the number you could satisfy is equal to the number of clauses). So what can we do in terms of approximation?

§23.2.1 A random solution

Just to get us in the mood, here's a very simple technique: try a random assignment. Would this be good or bad for a collection of OR clauses? For a clause with k variables, a random assignment is false with probability 2^{-k}. So this is great for big clauses — you get a very good approximation. Unfortunately, if you have a clause with only one variable, then it's just $\frac{1}{2}$. So if all your clauses are very big, then you'll satisfy a $(1 - 2^{-k})$-fraction, which is very close to 1; so that's a great approximation. But if you've got even one clause with only a single variable, then you can only get a 2-approximation.

§23.2.2 An integer linear program

So let's try to do better, using our powerful linear programming relaxation technique. How can we write a natural integer linear program for Max-SAT? We should probably have a variable per literal (we'll call the things in the original problem literals, to avoid confusing ourselves), which we'll make 0 or 1 for true and false.

And we'll also have a variable per clause, which we'd like to have be 1 if the clause is satisfied by the assignment.

Let's call these literal variables y_i, and the clause variables z_j. So then we want to maximize $\sum z_j$ (we're just counting the number of satisfied clauses). But what sort of constraint should we impose relating the z_j's to the y_i's? We only want z_j to be allowed to be 1 if some literal in that clause gets the appropriate assignment. So we require

$$z_j \leq \sum_{i \in C_j^+} y_i + \sum_{i \in C_j^-} (1 - y_i),$$

where C_j^+ represents the set of all the variables that are positive in the clause, and C_j^- all the variables that are negated in the clause (if any of the first variables are 1 or any of the negated variables are 0, then we're allowed to make z_j be 1). If we have the requirement that $y_i, z_j \in \{0, 1\}$, then this captures exactly what we want — you can only get your one point for the clause by satisfying one of the variables in the clause.

So we've made an ILP; and as usual, we then solve the fractional version of that, and that gives an interesting relaxation. We solve that, and get a fractional solution; that means instead of being in $\{0, 1\}$, all our y_i and z_j are numbers between 0 and 1.

§23.2.3 Randomized rounding

Now we need to round these numbers. How can I round a number between 0 and 1 in a way that kind of preserves its value usefully? How about we round it using some randomness, in a way that preserves its value in *expectation*? Maybe if I find some distribution where the *expected* values of the variables are equal to these numbers in the fractional solution, but they're integers — maybe that's close enough to the fractional solution to give us something.

This is a technique known as randomized rounding. We've got a variable between 0 and 1, and we want to round it to 0 or 1. So for each variable with value y, we set

$$\hat{y} = \begin{cases} 0 & \text{with probability } y \\ 1 & \text{otherwise.} \end{cases}$$

Now $\mathbb{E}[\hat{y}] = y$. And the reason expectation is nice is we have linear constraints, so we have linearity to help us — the expected value of a constraint will also be equal to the fractional value of the constraint.

This starts to get at why this is a general technique for rounding ILPs, or at least $\{0, 1\}$ ones. But it's not completely straightforward.

We can round the y_i in this way, and that's very appealing — the right-hand side of these constraints has exactly the right value in expectation. But why aren't we done? The problem is with the inequalities. We're rounding the y_i's, but what can we say about the z's? In a sense, these are what show up in our objective function — we want to maximize them. And z is just a representation of $\sum y_i + \sum(1 - y_i)$ but capped at 1 — so it's the maximum of this quantity and 1. And if we apply a randomized rounding step to the y's, we get some sort of distribution for z, but we don't know what it is. And in order to understand the outcome of our rounding step, we need to understand something about that distribution.

§23.2.4 The distribution of z_j

For the analysis, we'll define a funny series of numbers

$$\beta_k = 1 - \left(1 - \frac{1}{k}\right)^k.$$

(These are going to show up, so we'll define them ahead of time.) If we plug in $k = 1$, then we get 1; if you plug in $k = 2$ you get $\frac{3}{4}$; if you plug in $k = 3$ you get 0.704; and so on. The point is these numbers are slowly decreasing, and their limit is $1 - 1/e$.

So now what do we want to understand in order to analyze our thing? We're interested in $\mathbb{E}[\sum \widehat{z_j}]$ if we do this rounding — that tells us what sort of score we can hope to get in expectation. Of course, thanks to linearity of expectation, this is just $\sum \mathbb{E}[\widehat{z_j}]$. And $\widehat{z_j}$ is just a $\{0,1\}$-random variable, so this is just $\sum \mathbb{P}[\widehat{z_j} = 1]$.

Since it doesn't matter for the analysis, we'll assume all variables in clause j are positive (if we had a negated literal, we could swap it for its negation). In that case,

$$\mathbb{P}[\widehat{z_j} = 1] = 1 - \mathbb{P}[\widehat{z_j} = 0].$$

And the only way $\widehat{z_j}$ can be 0 is if all the $\widehat{y_i}$ in clause j are 0. And what's the probability that happens, if we're doing randomized rounding? It's $\prod(1 - y_i)$. So we get

$$\mathbb{P}[\widehat{z_j} = 1] = 1 - \prod_{i \in C_j^+} (1 - y_i).$$

For comparison, the LP gives a value of z_j to clause j. So in our linear program, it looks like clause j is contributing z_j; in our randomized rounding, it contributes this funny mess instead. How can we relate these two things? Well, if we can show some ratio between the funny quantity and z_j, just as we did in the facility location problem, then their sums are also close, which means the number of clauses we get is close to the fractional value.

So how can we relate these two quantities? Well, if we want to be pessimists, the worrisome thing is if we have a pretty large value of z_j (so it looks like we should be getting a big payoff), but $1 - \prod(1 - y_i)$ comes out to be very small, so we actually don't. So what we're worried about is, how small can this get? That'll determine the approximation ratio. And if we're worried about how small this can get, we're really asking how big the thing we're subtracting can get.

And how do we maximize $\prod(1 - y_i)$? We know from the linear program that $z = \sum y_i$; and that says all the y_i have a certain fixed sum. And if you have a fixed sum, the way to maximize this product is to set all of them to be equal. So we should set all the y_i's to be equal, and their value will be z_j/k, where k is the number of variables. In that case, what we get is

$$1 - \left(1 - \frac{z_j}{k}\right)^k,$$

as the worst possible outcome for the randomized rounding. And we want to compare this to z_j.

Claim 23.2 — We have

$$1 - \left(1 - \frac{z}{k}\right)^k \geq \beta_k z.$$

If that's true, it means that for clauses of k literals, we're getting a β_k-approximation. (This is not a precise statement, because we'll have clauses of lots of different sizes, and we have to think about how they combine.)

Proof. We'll prove this by picture. We have these two functions $\beta_k z$ and $1 - (1 - z/k)^k$, and we're interested in what happens as z ranges between 0 and 1. The linear function starts at 0 and rises to β_k. The question is, what happens to the other function (on the left)? Well, if we plug in $z = 0$, we get 0. And if we plug in $z = 1$, we get $1 - (1 - 1/k)^k = \beta_k$. So at the two ends of the interval, these two functions have the same value.

If we take the first derivative of the left function, we get

$$\frac{d}{dz}\left(1 - \left(1 - \frac{z}{k}\right)^k\right) = \left(1 - \frac{z}{k}\right)^{k-1},$$

which is equal to 1 at $z = 0$. All the β_k are smaller than 1; so this function starts out growing faster than the linear function.

The second derivative is

$$\left(1 - \frac{1}{k}\right)\left(1 - \frac{z}{k}\right)^{k-2},$$

which is less than 0. So this function is curving down, all the time. If it were curving down all the time and passed through the linear function somewhere in the middle, it would never be able to turn around and get back to the top. So the only thing that can be happening is that it always stays above. □

How good an approximation is this? That depends on k. But we did say β_k started with a large value and shrank to a limit of $1 - 1/e$, so none are smaller than that. This means we have a $(1 - 1/e)$-approximation for Max-SAT, which is better than a 2-approximation.

Here what we had was a very generic technique for rounding a relaxation of a linear program; but you still need creativity to analyze the result of that rounding. It mostly reduces your search space — it's a good thing to try first, because it's easier to analyze this than come up with an arbitrary rounding scheme.

§23.2.5 Combining the two approximations

The story isn't done — we had one approximation algorithm based on complete randomness that got a 2-approximation, and another that got a $(1-1/e)$-approximation. But they're actually kind of complementary. The first algorithm is great if your clauses are all very large, but does very poorly when your clauses have size 1 (it's those that give you the worst bound). The second is the other way around — it does great on clauses of size 1 (it's a 1-approximation algorithm — that's not surprising, because if your clauses have size 1, you don't have a product — everything is linear) — but as soon as you get bigger, the approximation ratio gets worse. So wouldn't it be cool if we could combine them in some way?

Of course we can — we can run them both, and take the better outcome. How well does this do? Well, we're going to analyze this in a pretty weird way — consider the *sum* of the outcomes of these two roundings. One of our two outcomes is going to be at least half of this sum; so it's enough to just analyze the sum. (We look at a sum because with randomization, linearity of expectation is powerful.)

For a clause of size k, the first method achieves $1 - 2^{-k}$ in expectation; and the second method achieves $\beta_k = 1 - (1 - 1/k)^k$ in expectation. So if we just write those out, on the top we've got $1/2$, $3/4$, $7/8$, ...; on the bottom we've got 1, $3/4$, 0.70, ... What we can see here is that the top starts at $1/2$ and gets bigger, the bottom starts at 1 and gets smaller; and they sort of cross at $3/4$. If you work out the math, this sum is always at least $3/2$, for any value of k; and that means in expectation, the sum of the two outcomes is going to be at least $3/2$ times the number of clauses; so one is going to give you at least $3/4$.

So you get a $3/4$-approximation, which is better than either of the ones we started with.

> **Student Question.** *Can you do better by weighting one of them by p and the other by $1 - p$?*
>
> **Answer.** Unfortunately not, because of the fact that they have a shared value of $\frac{3}{4}$ at clauses of size 2. There's actually a weird sense in which they're the same algorithm with slightly different thresholds — you can think of this as taking two variations and tuning between them.

Randomized rounding has been applied to a very wide range of problems, because it's so general.

§23.3 Fixed-parameter tractability

We'll finish up approximation algorithms today and start a new topic (to be decided) on Friday. We'll see one other trick developed for working with NP-hard problems. Instead of approximating them, we'll look at *fixed-parameter tractability*.

The idea is that when you talk about hardness of a problem, you're talking about all instances; what makes the problem hard is that some instances are hard. No one instance is hard — there's always a fast algorithm for that instance. So hardness comes from having a collection of instances.

The idea behind fixed-parameter tractability is maybe there's a natural way to stratify instances into an easier class and harder classes. Maybe the problem is hard in general, but here's this easy class, and more generally, here's this way the instances evolve from being easy to hard.

We'll see this by way of an example, but there's a whole rich theory of it.

Fixed-parameter tractability is the idea you take all instances and divide into classes by some natural parameter. And then you bound the hardness in terms of the parameter. This is very abstract, so let's make it concrete.

§23.4 Vertex-Cover and the bounded search tree method

Let's go back to the good old Vertex-Cover problem — it's NP-hard, hard to approximate, and so on.

> **Question 23.3.** What makes Vertex-Cover hard?

Let's assign a parameter k, which is the size of OPT.

We claim that if k is small, then the problem is easier.

Suppose the vertex cover only had size 1. How long would it take you to figure out the optimal vertex cover? Just n — we just check all the vertices, and see which one is the cover.

More generally, if we have k vertices in the cover, then we can find it by brute-force search — try all sets of vertices, and check which is optimum. That requires checking all the edges; so there is a simple mn^k brute force algorithm.

That's so simple it's not interesting (any NP-hard problem tends to have this n^k time bound, for k defined appropriately).

> **Question 23.4.** Can we get a better dependence on k?

We'll see a general technique, called the *bounded search tree method*. This is going to try to be clever about doing a brute force search through the space of possible solutions.

We take any edge of the graph. Then we know one of the endpoints of this edge must be in the vertex cover. So let's try both — we take one vertex, try putting it into our candidate vertex cover we're constructing. And for each, we just recurse to find the rest of the optimum vertex cover.

So we literally have this binary search tree — we try the left-side vertex, and then the right-side vertex. When we try a vertex, it covers some edges, so we throw those away; then we pick another edge that has survived, and do the same thing. So we're walking down this search tree.

How big is this search tree going to get? We know that the optimum solution involves k vertices; so if we guess correctly at every step, then k steps will suffice to find the optimum vertex cover. This means this is a depth-k degree-2 search tree. (You can use BFS or DFS, as long as you stop at depth k.) This means it only takes you 2^k steps. And doing a step requires going through edges to see what's covered. So you get a runtime of $m2^k$, which is dramatically better than mn^k — even if $k = \log n$, you still have a polynomial-time algorithm.

§23.5 The kernel method

That's one trick for dealing with a small fixed parameter; now we'll see an even cooler trick, called a *kernel* method (though it has nothing to do with machine learning).

The idea is to, in polynomial time independent of k, find a small difficult 'kernel' of the problem, and then solve it by brute force. Again, this is quite abstract, but it will be quite clear once we see an example with vertex-cover.

Again, we assume a vertex cover of size k.

> **Claim 23.5 —** If the optimum vertex cover has size k, then any vertex of degree greater than k must be in OPT.

Proof. If it wasn't, then you'd need the edge of every end coming out of it to be in OPT — all its neighbors would have to be in OPT — and there's too many. □

So we can just take these vertices right away; and that reduces us to a graph of degree at most k. Now we might be looking for a vertex cover of size smaller than k (since we already took some vertices), and we need to find the rest; but we need to find them in a graph with degree at most k.

How many edges can this graph have? Let's think about that optimum vertex cover. We have a vertex cover of size k; all edges are covered by that vertex cover. But each vertex in it covers only k edges, which tells us that in total, there are only k^2 edges in the graph.

So after we take out the obvious stuff — which takes $O(m)$ time — we're left with a graph that only has k^2 edges. Now we can use the bounded search tree method, but where m is bounded by k^2; so bounded search now gives a runtime of $2^k k^2$. This means overall, we get a runtime of $O(m + 2^k k^2)$. The bounded search tree method took us down from a base of n to a constant, but it was multiplicative. Kernelization rally separates you into a problem where you have a polynomial independent of k, and another one that's independent of the size of the problem and depends only on the parameter. (So if you think about the parameter as a constant, it doesn't multiply on the size of the problem.)

§23.6 Fixed-parameter tractability theory

There's a whole theory of fixed-parameter tractability. Some problems are fixed-parameter tractable, like vertex-cover. Other problems, like clique or independent set, are the opposite. Independent set is $W(1)$-hard, meaning that if you could find a FPT algorithm for independent set, it would give you one for all sorts of NP-hard problems — so it's essentially the hardest problem for fixed-parameter tractability. If the independent set has size k, we only know how to solve it in n^k; we don't know how to make a constant base.

(In general, you want something that's allowed to not be polynomial in k, but it can't involve the other size parameters — the k parts need to be separate from the other stuff. But there's a bizarre theorem saying if you can get something like $2^k \cdot m$ — a multiplicative factor — then you can also get something like $2^k + m$, where it's additive.)

Another example of a parameter people care about is *treewidth* (which is used a lot in machine learning).

§24 November 8, 2024

§24.1 Online algorithms

Today we're starting a new topic, called online algorithms. Until now, we've looked at problems where you're presented with input, and you run an algorithm and produce output. Online algorithms is a study of what to do where you have to provide an answer *while* the input is arriving — like building the engine while the airplane is flying.

In the real world, you've often got an ongoing process that keeps giving you updated information, and you have to react to that in an effective way, keeping in mind that newer information is going to arrive later that may change what you should do, but you don't have it yet. So you're simultaneously thinking about what to do now and planning for the future.

We saw this to some extent with data structures — those are sort of online algorithms responding to requests for insertions and deletions and so on. Now we'll look at a broader class.

§24.2 Setup and definitions

Formally, we have an input sequence σ (of symbols or values or states). As each symbol arrives, you have to act — you have to produce an output, update something, and so on. All of these actions you take come with costs. So over the entire sequence, you pay a total cost $c(\sigma)$. And your goal, since we're talking about costs, is that you want to minimize it. (This is the cost of your algorithm on the sequence, so we could really write $c_{\mathcal{A}}(\sigma)$.) You want to minimize this (you could also imagine maximizing profits, where the story is similar).

The problem is that you don't know what's going to happen in the future. So how do you do the right thing? That's one question, but also, how do you decide if you're doing well or not? There are certain input sequences where it's simply impossible to do well. If you're investing in the stock market (a good model of online algorithms) and the stock market just goes down and down and down, there's just no way to make a profit. Another place — the problem that kind of led to the definition of the model — is paging in computers, or caching. You've got a cache and you're reading things out of memory, so things get pulled into the cache where they can be read quickly again. Eventually the cache gets full, and you have to evict something. You can try to be clever about what you're evicting. If the program has good locality of reference, you might be able to do a good job. But if each request is for a new piece of data, you simply can't take advantage of the cache.

What this highlights is you have to compare your performance to a reasonable competitor (we can't just say a high cost is bad, because sometimes every algorithm has high cost). So we want to compare our cost to the optimum algorithm for each sequence — we want an algorithm that does well when it's *possible* for some algorithm to do well (if no algorithm does well, it's fine for our algorithm to do poorly, because it has to).

To formalize this, how do we measure the comparison between our algorithm and the optimum? We want some number that compares these performances, per sequence. In approximation algorithms, the first thing we talked about was additive approximations; we could ask your cost to be within an additive amount of the optimum for every sequence, i.e.,

$$c_{\mathcal{A}}(\sigma) \leq c_{\mathsf{OPT}}(\sigma) + k.$$

This is usually not possible, because these sequences can grow infinitely long — so if your algorithm makes mistakes periodically, its divergence from optimum will grow additively large. So instead, as with approximation algorithms, we focus on a multiplicative approximation.

Definition 24.1. An algorithm $\mathcal{A}$ is k-competitive if for every sequence σ, we have

$$c_{\mathcal{A}}(\sigma) \leq k \cdot c_{\mathsf{OPT}}(\sigma).$$

What's important to realize is that when we consider $c_{\mathsf{OPT}}(\sigma)$, we're considering *all* algorithms, so we can think about the algorithm that's the best for this one σ. That means this algorithm is allowed to know the future — we're running this algorithm knowing what σ is. That's what makes this challenging — we want an algorithm that's good *in hindsight*. And in hindsight, you know everything that's happened, so you might be able to design a much better algorithm. So how can you design an algorithm that's good, without knowing that?

Remark 24.2. Sometimes we also allow an additive constant term — i.e., $c_{\mathcal{A}}(\sigma) \leq kc_{\mathsf{OPT}}(\sigma) + O(1)$ — to pay for startup costs. This is called *asymptotically k-competitive*.

Definition 24.3. We call k the competitive ratio of our algorithm.

We're looking for algorithms with small competitive ratios — ones that look good even in hindsight.

§24.3 Ski rental

Here's a very classic example.

David did a bit of skiing in college (3 times) and then stopped. In hindsight, it's a good thing he didn't buy skis, becaues that would have been a massive investment he didn't make use of; instead he rented skis, which was the right decision in hindsight.

In this problem, more generally:

Example 24.4

You repeatedly go skiing. Each time, you can rent skis for 1 dollar (this is a very old problem), or you can buy them for T dollars (with $T > 1$). What should you do?

First, what's the space of possible algorithms? You rent for some amount of time, and after some time you buy; and once you buy skis, you never need to rent again (unless you forget them at home). So you rent for a while (d days) and then buy. So then the optimization opportunity is to think about, how do we choose d? Again, we're looking at this from the competitive perspective — we want to worry about hindsight.

Let's look at some candidates. First suppose we buy right away. Would that have good competitive ratio? We're thinking about competitive ratios, so we're asking about the worst possible future (we look at the sequence that gives us the worst ratio to the optimum). And if we buy skis right away, the sequence where we go skiing exactly once and never again gives a ratio of T.

What if we rent all the time? That has infinite competitive ratio, because if we never buy skis and just rent forever, that means in hindsight I should have bought skis right away — the optimum cost on the sequence of skiing forever is T, while I'm paying ∞. So this is not competitive at all.

So neither extreme makes sense. But is there somewhere in the middle that's more sensible?

We want to put on an adversarial mindset — if we choose some algorithm, the adversary picks the sequence that makes the worst possible competitive ratio for this algorithm. If I rent for d days and then buy, what sequence gives the worst possible ratio?

Claim 24.5 — The worst sequence is to do $d + 1$ days of skiing.

In other words, the moment you buy your skis, that's when you break your leg. Why is that the worst outcome? If you keep skiing after you buy your skis, it doesn't cost you anything more; from the adversarial perspective, all that could happen is it could cost the adversary more. So the adversary has no motivation to make the sequence longer.

More precisely, before I buy skis, the ratio is 1 if the adversary is also renting, and keeps getting worse if the adversary bought their skis. So before I buy skis, the ratio is not improving. Meanwhile, after I buy, my cost doesn't increase, whereas the adversary's can only increase; this means the ratio is not getting worse. So that means the moment with the worst ratio is the moment just after I bought the skis.

If we look at the purchase moment (where I bought the skis), I pay $d+T$. What does the adversary (or OPT) pay? They pay $\min\{d+1, T\}$ — either they rent every day or they buy at the start (whichever is better). So our competitive ratio is

$$\frac{d+T}{\min\{d+1,T\}}.$$

We want to pick d to optimize this; this is best at $d+1=T$ (up to there it shrinks, and past there it grows); this means we get

$$\frac{2T-1}{T} = 2 - \frac{1}{T} \leq 2.$$

So this simple algorithm of 'rent until you've spent as much as the price of the skis, and then buy' is 2-competitive to the optimum algorithm.

§24.4 Market

Let's do a few more financial examples. We'll look at a very simple model of a market. I have something I want to sell; the price of it goes up and down. How much of it should I sell, and when?

Example 24.6

Suppose I have 1 (infinitely divisible) unit of boba, and the price fluctuates; I want to know how much I should sell and when.

The way we've formulated it, there's really no good answer. Thinking adversarially in this model, how do you make me incredibly depressed in hindsight? Right after you sell, the price skyrockets. (Imagine it's pita, I sell the pita, and then Clover closes and there's no pita in town and the price of pita goes through the roof.)

So we can't solve this problem; it's too unconstrained. So let's add some constraints. Suppose I know a maximum price M and a minimum price m. Now how well can I do?

We could try to wait until the maximum price is reached, and then sell everything. But this raises another technical problem — if you sit around waiting forever and the market closes, then you haven't made any money at all. That would force you to sell on the first day, unless you also know the final option to sell. (If things end with some stuff unsold, it all gets sold at this minimum price of m.)

There's an obvious competitive ratio you can achieve — if you sell right away, you'll earn m, while the adversary makes at most M. So the competitive ratio is just the ratio M/m, which we denote by Φ.

But can we do better? This time, the space of algorithms isn't about how many days you wait, but at what price you'll do various things (because if we talk about e.g. waiting for 3 days, then the adversary can also freeze the market for 3 days).

If we want to do better, we don't just want to sell everything on the first day. We could try waiting until it reaches M and selling everything, but the adversary will then just never take the price to M. But the adversary also wants lots of money themselves (in order to say they got much more than us), so what they'll do so that they get a lot of money and we don't is to bring the price up to $M-\varepsilon$; and then they sell everything. Meanwhile, we'll be waiting for that wonderful day when the price reaches M, which never happens; so we'll end up with just m.

So we can't do that.

One idea is that we can set a *threshold* (or *reserve price*, if you've taken economics), which we call R. If the price rises above R, then we sell everything. Otherwise, we wait.

What's the outcome going to be? Well, we're going to get R if the price rises above R, and m if it never does. So what's the adversary going to do, for a given R? Well, they have two strategies. The adversary

knows our algorithm, so they'll deliberately design the sequence to mess with it. The idea is they just need to decide if they're going to pass the reserve price or not. If they do, I get R; they might as well get as much money by going to M. If they decide not to pass it, then I'm going to get m; but they want as much money as possible given that they can't cross R, so they go up to $R - \varepsilon$.

So the adversary can get a ratio which is the worse of $\frac{M}{R}$ and $\frac{R-\varepsilon}{m}$. The adversary will pick whichever of these is worse. So if we want to do the best possible, we'll pick R so that these two quantities are equal. So we can balance at $R = \sqrt{Mm}$, and this gives a ratio of $\sqrt{M/m}$.

This is not great, but it's better by a significant amount than the naive algorithm.

Can we do better? Well, if you're trying to sell a car (a thing that you have 1 of), then you can't really do better. (We'll revise that statement in a minute, but roughly speaking this is true.) But boba is a liquid, so you can divide it up. Is there a way to take advantage of the infinite divisibility of this liquid? You could imagine we break the boba into multiple containers, and put different reserve prices on different containers. We can broaden the range of algorithms beyond a single sell point — instead, for each new high, we may sell a certain additional amount of what we have. This will look sort of like a cumulative distribution between 0 and 1 — how much we sell at each price between m and M.

So then the question is, how do you set that dependent selling function — what should it be? Let's sort of creep up on this.

Suppose I made two bundles of stuff, instead of one; now I can think about having two different reserve prices. Originally, we were able to achieve a square-root ratio by putting a reserve price right in the middle. What if I have two separate things, so I make two separate reserve prices? When we're fighting with the adversary, we're worried that the adversary is going to go right below our reserve price, or blow through it and we only get the reserve price. But now we have two different reserve prices — we have m and M, and two reserve prices. And we sell some stuff if we pass the first reserve price, and some if we pass the second. In TCS¡ the fact that each of these is half basically doesn't matter (it's just a constant factor). So how do we hedge our bets — how do we keep the ratios small regardless of where the adversary ends up? If they pick something in the first interval, we get m and they get R_1. If they pick something in the middle, we sell half of the thing at our first reserve price, and they can get to R_1. And so on. So the ratios that matter are R_1/m, R_2/R_1, and M/R_2. And how do we get all those to be equal? We use cube roots — we set $R_1 = M^{1/3}m^{2/3}$ and $R_2 = M^{2/3}m^{1/3}$. This is a generalization of the geometric ratio we took before; and then we achieve a $(M/m)^{1/3}$ ratio.

If we think about constant factors, m vs. $2m$ is a constant factor we don't care about, and similarly with $2m$ and $4m$. So going a bit further, we'll set reserve prices at powers of two — we'll have $\log \varphi$ reserve prices at $2^i m$ ($\varphi = M/m$, so $\log \varphi$ is the number of powers of two we can fit). And we sell a $1/(\log \varphi)$ fraction at the ith reserve price.

Now the argument is pretty straightforward — I will sell *something* for at least half of the maximum price (which is the price the adversary sells at). And I'm going to sell at least a $1/(\log \varphi)$ fraction of my stuff at half the maximum possible price. This means I get a $O(\log \varphi)$ competitive ratio.

> **Remark 24.7.** You can improve this a little bit by using slightly different ratios, and accounting for that if you're selling at a high reserve price, you also already sold stuff at lower reserve prices. This helps a little bit, but $O(\log \varphi)$ is asymptotically about as well as you can do.

Now let's go back to selling a car. This division of your product into multiple pieces gave us an exponential improvement, from $\sqrt{\varphi}$ to $\log \varphi$. Are we stuck at $\sqrt{\varphi}$ for the car?

Well, if you'll use a deterministic algorithm, it's easy to prove you can't beat $\sqrt{\varphi}$ — there'll be some price where you sell, and $\sqrt{M/m}$ is the optimal price to do that. But you can use randomization to effectively divide up the car into pieces, without actually dividing the car into pieces (which would kind of lower its value).

So for a car, what we can do is pick one of the above thresholds at *random*. Then with probability $1/(\log \varphi)$, you pick the one just below the maximum achieved price. So you receive at least half the maximum price with probability $1/(\log \varphi)$. And so in expectation, you are making $O(1/\log \varphi)$ of the optimum.

You can decide whether that makes sense when talking about a car — do you really just care about the expectation, or are you worried about getting nothing at all (which is what will quite likely happen)? What this points to is the fact that randomization is a way out of this bind you're in that the adversary knows what you're doing. It's important in the model that the adversary is *not* aware of your random choices — the adversary knows you have an algorithm that'll pick a threshold uniformly among these. But they don't know what random choice you made, but they can't make a sequence that's bad for your random choice. That gives you some unpredictability against the adversary that lets you significantly improve performance.

If you're selling only one car, then you might worry about the risk. But if you're running this many times, the expectation is the right thing to optimize, because you'll converge to the expectation over many runs.

§24.5 $P \parallel C_{\max}$ online

These are kind of toy problems; we'll now mention a less toy problem that we have to worry about pretty frequently.

We looked at $P \parallel C_{\max}$ as an approximation problem — here are some jobs, schedule them. There's a natural online version — jobs keep arriving and you have to assign jobs to machines as they arrive. We compare that to the optimal algorihtm, which knew what jobs were going to arrive and could make an optimal assignment. You can't do that because you don't even know what is the last job — you just keep getting jobs, and you have to assign them.

What's a good algorithm for this? That's kind of a trick question — Graham's rule, putting the job on the machine with the least work, is an online algorithm. Our analysis shows that within each moment of scheduling a job, you are within a factor of 2 of OPT; this means we are 2-competitive.

What's interesting is you can apply the online algorithms framework even to problems that are NP-hard — we don't even know what OPT is, but we know we have a competitive ratio of 2.

On the other hand, the other algorithms we studied (we had this intuition that if you schedule the largest jobs first it improves things; that gives 4/3; and then we used grouping and rounding to get a PAS with enumeration) — none of that works online, because you need to have all the jobs to decide the groups.

For quite a long time, the question remained open, can you beat 2 (or really $2 - 1/m$) in the online case?

There was an exciting moment when the answer to this turned out to be yes — Bartal, Fiat, and a number of other people, in 1995, had a breakthrough where they improved the competitive ratio to $2 - 1/70$. Yes, it's amusing this is called a breakthrough, but it was because no one had any idea how to improve 2.

The idea is, if we think back to looking at Graham's rule, the bad case for Graham's rule was when a ton of tiny jobs arrived and you spread them out, and then a gigantic job arrived. So you had all these equally balanced machines, and the gigantic job had to land on one of them and that gave you a factor of 2 error.

The trick is to prevent that from happening. They intentionally underload some of the machines. By keeping some machine sunderloaded, they ensure that if giant jobs arrive, they'll be able to put it onto a machine with slightly less than the average load; so the new load of their algorithm will be slightly less than twice the average.

If you only kept one machine underloaded, then the adversary would send you two giant jobs; the first would go on that machine, and the second on one with the average load. So it's not enough to just keep one. Instead, they keep a constant fraction. This is good because if the adversary sends enough giant jobs to fill all of those, then they've pushed up the average laod, so now the *other* machines are below average. So there's a clever juggling act in keeping this balance so that you can use your less loaded machines for really big jobs if they arrive.

This technique went through a series of refinements. In 1996, David and a bunch of other grad students improved this from 1.986 to 1.945 (by enumerating the entire state space of possible distributions of loads, and running a big linear program to show you could always keep within the good space and never be pushed into the bad ones). Then in 1997, Albers improved this to 1.923. There's a lower bound of 4/3 or 3/2 or something; they're not tight. Also, these latter two algorithms are kind of ugly; David believes the optimum algorithm will have beauty to it and these don't. But they all share the same idea of maintaining an unbalanced distribution to handle large jobs.

Next, we'll see a more serious complicated problem and analysis.

Student Question. *What's the runtime?*

Answer. We've said nothing about the runtime of any of these algorithms. In fact, when you do competitive analysis, people don't worry too much about the runtime, just the competitive ratio. If you can get a good one (possibly by exploring an exponential space of options), then you can add on that it would be nice if the runtime were polynomial. Some papers on online algorithms say there was a competitive ratio, and we showed it can be achieved in polynomial time. But for starters you don't worry about it. (Of course you want it to be finite, but it's hard to make algorithms that aren't.)

§24.6 Paging

This is the problem of dealing with slow memories on a computer. You have a computer. In the traditional model of paging, you have a fast small 'memory' and a slow large 'disk drive.' So you can access things very quickly in memory, but you don't have a lot of it; most of your work is going to have to be stored on the slow large disk drive. Because the disk drive is large and slow, going to the disk drive to read a single value is wasting a lot of time. So the disk drive is in blocks of data, and you can read one at cost 1.

The problem σ is a sequence of data accesses. For this problem, we're not in control of where the data is; each piece of data is in some block somewhere on the disk. If the data's block is in the memory, we call that a *hit*, and the access is free. IF it's not in memory, we call that a *miss*, and we have to pay 1 to fetch the block (in order to serve that data access).

The first time we get a request for data, we'll have to fetch it off; but after that we can just keep it in memory and make the remaining accesses free. What goes wrong? The problem is that memory is small. If the memory gets full of blocks, then on a fetch, you must *evict* a block to make room for the new one. To quantify this, we'll say that memory holds k blocks of data from the disk.

Our cost function is just the number of misses that occur as a result of our strategy.

What's our strategy? The flexibility is in deciding which block we evict to make room for a new block we read.

We described this in terms of a memory and disk drive, but in modern computers you have a whole memory hierarchy — every processor has a small cache on the processor, and it's main memory that's slow (it takes many processor cycles to read data from main memory), so they read it in blocks and store that in their local cache. In fact, the registers of the processors are in a sense a really small cache. You also may have multiple levels of caches; the bigger they get the slower they get. So you don't want to access the big caches; you want your data in smaller caches. So at each level, you want a decision: when my cache is full and I pull something else in, what do I evict to make room for the new block?

Remark 24.8 (Logistics)**.** Monday is a vacation. On Wednesday David will be at a conference, so we will have a video lecture, which will be a completion of online algorithms. Looking ahead, psets are going to end before Thanksgiving (the last is due Wednesday just before Thanksgiving). Psets will be normal in every sense. One will have a requirement to submit a 1-paragraph description of a project. We'll give you information about what they can be, you'll submit a description, and you'll get feedback about whether the project is suitable for the class. Suitability means it should cover the kinds of algorithms we study in this class — not the specific algorithms, but there's a focus on combinatorial algorithms (not numerical or number-theoretic or cryptography); we should have a good sense of what sorts of problems fit into this. Online algorithms are fine; next week we'll see computational geometry, which is also fine; parallel algorithms will be too, though we won't have discussed them by the time of the projects.

After Thanksgiving there will be optional lectures (probably not on Fridays, but the Monday-Wednesday ones will continue as long as people want), but all we have to do is get projects done. The second-to-last lecture will be a peer editing session, where you bring your project — which has to be done — and we swap them around and give each other feedback. They're due the final day of class (since we're not allowed to make things due later).

What sorts of strategies make sense for this? I've got a full cache and a request; which block should I evict?

One strategy is the *least recently used* (LRU) strategy. This is also the intuition behind splay trees. If you haven't accessed something for a long time, maybe your algorithm is done with it, so you can let it go. Similarly, if something was recently accessed, maybe you're in the middle of working with it, so you don't want to evict it.

Another strategy is FIFO (evict whichever page came in first). This is similar, but you ignore if you've looked at it again.

There's also *flush when full* — keep putting stuff in the cache, and when it's full, you toss everything and start over (which can sometimes be easier to implement).

There's also LIFO — evict the most recently fetched thing.

There's also *least frequently used*, which also intuitively makes a lot of sense — if you don't use a page very often, toss it. This takes a bit more work, because you have to count how many times each page (or block) gets fetched.

The question is, can we apply analytic techniques to understand theoretically how well they do?

This is actually exactly why competitive analysis was invented — this was the original paper. Why? If you think about accessing new pages all the time, no caching algorithm can do well. So you need to compare your performance against the best possible — what the adversary does knowing your algorithm. Cycling through $k+1$ pages is a very good adversarial strategy against some of these algorithms, as we will see soon.

It turns out analytically that LRU, FIFO, and flush when full are all k-competitive. So there is a decent competitive ratio. LIFO and least frequently used are not — you can make them do infinitely worse than the optimum.

And k (achieved by the first three) is the best possible for deterministic algorithms. (You can guess where that qualification leads.)

We'll see if we can prove that LRU, FIFO, and flush when full are k-competitive. The easy argument is to show they're $k+1$-competitive. For this, we consider our sequence σ and break it into intervals of $k+1$ faults — we start a phase, there's a fault at some point, then there's another, and so on. On the $(k+1)$st fault, I say the phase is over, and start looking at the next phase.

We'll compare ourselves to the optimal algorithm phase by phase. If there are in fact $k+1$ *distinct* faults, what does that tell us about the request sequence, if we're using LRU or flush when full or FIFO? What do you have to do to make LRU fault more than k times?

Claim 24.9 — If there are only k pages in a (sub)sequence, then you only get k faults.

Proof. The first fault that you get, now it's the most recently used page. So it's going to stay in memory under LRU (or FIFO or flush when full) until k other pages come in and make it no longer one of the k most recent pages. □

Turning this around, $k+1$ faults implies you have $k+1$ distinct page requests. And if I got $k+1$ distinct page requests, the optimum algorithm has to have had at least one fault — OPT can't possibly have all of them in memory, so one of them is a fault.

So we've got these phases; in each I make $k+1$ faults, but the adversary made a fault. So if we sum over phases, the adversary makes one fault for each $k+1$ that LRU makes. And the same argument applies to FIFO or flush when full.

In fact, the correct answer is that they're k-competitive. The only reason that we care about k vs. $(k+1)$ is that k is tight — you can't do better than k, so it's nice to have an algorithm that does exactly k. Proving that just requires a bit of extra accounting of what happens between phases. This analysis is a bit sloppy because the $(k+1)$st fault you're using to declare the adversary has a fault — you can sort of think of this as the start of the next phase instead, so effectively you only have k faults per phase instead of $k+1$. This will be part of Wednesday's video lecture; we'll also see how to do even better with randomization (and it's a *lot* better — you can actually get $\log k$, which is an exponential improvement, by flipping coins to prevent the adversary from knowing exactly what you have in memory). The adversary is going to be requesting pages you don't have; if they don't know what pages you have, that becomes difficult.

§25 November 13, 2024

§25.1 Paging

We've gone through some simple problems for online algorithms and David started introducing paging, and now we're going to do an analysis of the algorithms we have for it.

§25.1.1 Proving $(k+1)$-competitivity

We'll start with an easy analysis, where we'll just prove that we're $(k+1)$-competitive. What we'll do is we'll consider our request sequence σ. And we're going to break it into maximal *phases* of $k+1$ faults by our algorithm. We'll do the analysis for LRU, but you can also do this for FIFO.

So we consider any sequence and look at what LRU does, and on every $(k+1)$st fault we declare the end of a phase and start looking at a new phase. If there are $k+1$ distinct faults, what can we say about this phase?

Claim 25.1 — If there are only k pages requested, then you can't have $k+1$ faults.

This is because if there's only k pages requested, none are evicted; so we have at most one fault per page. So if there are only k pages in the phase, then there are at most k faults. And the phase has $k+1$ faults, so there has to be more than k pages requested.

And if more than k pages are requested in a phase, what can we say about OPT? There has to be at least one eviction — it might be that k of them were already in the cache, but at least one definitely wasn't. There's only k spaces in the memory, so OPT must miss at least once.

So every time we see $k+1$ faults of LRU, we can associate that with one fault in that phase in OPT. And so we have our $(k+1)$-competitive analysis.

§25.1.2 Improving $k+1$ to k

Now, if we refine this analysis a little bit, we can actually improve the competitive ratio to k. We might not give all the details, but let's get started. Instead of breaking into phases of $k+1$ faults, we'll break into phases of k faults — this is the natural thing to do if we want to prove a k-competitive ratio (we want to say for every k faults of our algorithm OPT faults once). But the problem is if there's only k faults, what if OPT had all those in memory and didn't fault on any of them? So how do we get one fault into OPT?

Notice that a phase *begins* with a fault — it basically begins with the $(k+1)$st fault, the first request that can't fit into the previous phase.

Let q be the last page requested before the current phase. (We basically have to do some more careful analysis to find a page to make OPT fault on, and that's what we're going to do.) Note that q is initially in memory; that'll be our source of blame.

Case 1 (Our algorithm (LRU) faults on q in this phase). Why would LRU fault on q? Well, q is the most recently used at the *beginning* of the phase, because we just asked for it. The only way to make a fault happen is to evict q, and for that you need k pages that are more recent then q; that means you need k requests in this phase which are *not* q. That means we have k requests here, along with the request for q from the previous phase; that's $k+1$ distinct pages, which means OPT faults on one of them. But it faults during *this* phase (we're finding a fault that happens in this phase by OPT, not before).

Case 2 (LRU faults on a different page, but it does so twice). Again, for that to happen, there must be k other requests between those two faults, so $k+1$ in total; so OPT must fault on one of those as well.

Case 3 (There is no fault on q or twice on any page). That means q is in the cache at the start, and stays there — it's never evicted. This means the k faults that happen in the phase must be for other pages. And that again gives us $k+1$ distinct requests in total, so we conclude also in this case that OPT must fault.

These are the three cases; we've shown that in each, even though we're only looking at k faults in a phase, in every case OPT has at least one fault in the phase. So every k faults in LRU corresponds to one in OPT.

§25.1.3 Tightness of k-competitivity

Why do we care about k vs. $k+1$? The big deal is we can actually show that k is the best possible result — no algorithm is better than k-competitive. We do that by designing an adversary — we're given some algorithm, and we're going to generate a sequence with a bad ratio. This is actually pretty easy. We'll only use $k+1$ distinct pages. The adversarial sequence is that each request is for the page that our algorithm doesn't have. This means our algorithm faults on every step.

It's easy to make request sequences where the algorithm faults on every step; but the point is that with this sequence, we can make a prophetic algorithm that does well. What does it do? Initially it's empty and has faults, but its cache is going to get populated. What should we evict?

We'll play a nasty trick — we know the sequence, so we'll evict the page that's *not* requested in the next k requests (including this faulting page). We make the bad sequence *before* we design the optimum algorithm. So we look at this sequence, and build our algorithm based on this sequence — when there's a fault, I look at the next k requests, and evict the page that's not asked for in the next k steps. That means my algorithm only faults every k steps. So we've just shown that the proposed algorithm cannot do better than a factor of k — there's at least one sequence where it's off by a factor of k. That means the factor-of-k bound is tight — and LRU and FIFO are both k-competitive.

§25.1.4 Final remarks

We might think k-competitivity is a disappointing bound. But in *resource augmentation*, we give the online algorithm $h > k$ space, where the optimum only has k space. It turns out that then LRU is actually

$\frac{h}{h-k+1}$-competitive (you can generalize the proof from earlier). So we showed that when $h = k$ you're just k-competitive; but if you give your online algorithm twice as much space, then you end up with a 2-competitive algorithm. So this is 2-competitive against an offline algorithm with half as much space, which is reasonable (you're trading a bit of space with the ability to see the future).

This bad case required the adversary looking carefully at what our algorithm did to design a worst-case. An analog would be in poker, getting to look at your opponent's cards before making your bid. If we want to overcome this, we have to hide things from the adversary. We'll see how to do this with randomization — if we make our algorithm unpredictable, then the adversary can't design these bad sequences, and we can get a significantly better ratio.

§25.2 Randomized competitive algorithms

To get past this weak bound of k, we'll introduce randomization. The problem with our deterministic competitive algorithm is that the adversary could predict exactly what it was going to do, and build a bad sequence by requesting the page that it did not have in memory (with only $k+1$ pages). We're going to now explore introducing randomization — then the adversary is not aware of the state of the online algorithm, so they can't make things as bad for the online algorithm as when they were.

We do have to be a bit careful in our definitions. So we'll first talk about the definition of k-competitiveness in a randomized algorithm.

> **Definition 25.2.** We say an algorithm $\mathcal{A}$ is asymptotically k-competitive if for every input sequence σ, we have $\mathbb{E}[C_{\mathcal{A}}(\sigma)] \leq k \cdot \mathsf{OPT} + O(1)$.

(The 'asymptotic' bit corresponds to $O(1)$.) Note that in the optimal algorithm, there's no point to randomizing — it has full knowledge of the future, so it can just do the best thing.

§25.3 Types of adversaries

We do need to be a bit precise about what kind of adversary we talk about. Our online algorithm is flipping coins to decide what to do; how much does the adversary know about these coin flips?

We can talk about an *oblivious adversary*, where the adversary has to give the whole sequence before I run my algorithm — so the order is that the adversary defines a particular input sequence, then I run my algorithm with all its random choices, and we compare the cost of my algorithm to what the adversary did.

For contrast, we can have a *fully adaptive adversary*, where the adversary knows my coin-flips before it gives the sequence. I'm still randomizing, but the adversary knows what I do.

If the adversary knows your coin-flips, then from their perspective it's not randomized at all (the choice of coin flips makes the algorithm deterministic). So there's not much you can say about this kind.

In the middle, there's an *adaptive adversary*, which knows your *past* coinflips but not your future ones — it understands what your current state is (because it knows all the coin flips you made in the past), but doesn't know what you'll do next. This is a reasonable model of certain kinds of systems. In our paging problem, for example, the choice you make about what pages to bring in affects how quickly your algorithm runs — it might have to wait while pages are being fetched. If you have a computer that's multiprocessing, this wait may lead to other programs being executed instead. This can actually sort of change the makeup of memory and can change when your program runs. Basically, your past choices, even if random, can affect what happens next in your program — if your program is a real-time program basing its action on now, then when it runs may affect what it does. And if it's using a paging strategy, that may affect when it's run. So there is this dependency; and it's reasonable to argue that in the worst case you may be affected

by random choices made in the past. It's not really reasonable to posit the adversary is adapting to choices you haven't made yet.

Unfortunately, even in this model, the k-competitive lower bound still holds. However, we are going to show how we can do much better against an oblivious adversary — if the input sequence has to be specified first, we'll see how a randomized algorithm can significantly outperform the deterministic ones.

§25.4 Marking algorithm for paging

What would be a natural randomized paging algorithm? You get a fault; what should you evict?

One natural idea is to evict a *random* page. That actually turns out to not be a bad idea; random is k-competitive, so it's not worse than the deterministic algorithms. But it's also not better.

One nice thing about it is that unlike the deterministic algorithms, random doesn't require any space to manage the algorithm — LRU or FIFO requires some extra space to remember when pages were accessed, and random doesn't care. So it has a slight benefit, but it's not really significant. We need to be smarter than this to beat k.

That comes from trying to incorporate some of the ideas of the deterministic algorithms. The deterministic algorithms recognize that you should not get rid of things that have been used recently. So we're going to add that to random.

This is an algorihtm known as the *marking algorithm*, due to Fiat et. al. from way back when. We'll initially have all pages marked (this is just a technical condition to make the analysis clean). Now what do we do on a fault? Well, if all the pages are marked, then we unmark all of them. Once you're past this step, there are some unmarked pages.

Now you evict a random *unmarked* page.

Finally, whether there was a fault or not, you mark the requested page.

What's the effect? When something is requested it gets marked, which means it's not eligible to be evicted. And it maintains that immunity from eviction until all the pages are marked, at which point everything gets unmarked and all the pages become eligible for eviction again.

This is the algorithm we're going to analyze. For the analysis, we're going to talk about phases; we're going to analyze each phase separately.

The first phase starts on the first fault, and it ends on the $(k+1)$st distinct request. This means a phase has exactly k distinct requests in it (it ends when you get the $(k+1)$st).

We'll analyze the cost of our algorithm as the sum of costs of various phases. Notice that these phases are defined without mentioning the marks — they're just a function of the input sequence, and not of our algorithm. But we claim they're very connected to the marking algorithm. How do these phases relate to the marking algorithm? Through a phase you'll slowly mark everything (because all the requests are distinct), evicting things that were unmarked; then eventually all the things in the cache are marked, and you'll unmark everything for the start of the next phase.

So by induction, we unmark everything exactly at the end of a phase. We start a phase with everything unmarked. Then we start marking things as they're requested. Once there are k distinct requests, we have k distinct things marked. So on the $(k+1)$st, we'll be in the situation where there are $k+1$ things marked; so we unmark them all, and that corresponds to the beginning of the next phase.

So even though these phases are defined just in terms of the input, they match exactly what the algorithm is doing.

Also by induction, at the end of a phase, our memory contains exactly the k pages of the phase — when something is requested it gets marked, and once it's marked it can't be evicted, so at the end of the phase you have exactly the k pages that were requested and marked.

Now that we understand these phases, we'll simplify.

Claim 25.3 — Without loss of generality, we can ignore all but the first request for a page in the phase.

In fact, we'll simplify our request sequence by getting rid of the extra requests for any page in a phase. We claim this only makes our competitive ratio worse — so if we're just trying to get a bound on the competitive ratio, this is a valid simplification.

Let's think about why that is. What can we say about second and later requests for a page in a phase? If you mark the page, then it's never going to get evicted (until the end of this phase), so it'll remain in the memory and you won't fault on it. So the marking algorithm will never fault on the second or later request for a page — they don't cost the online algorithm anything. They might cost OPT, but that only improves the competitive ratio — so taking them away makes things better for OPT without making things any better for the online one.

This means the phase has exactly k requests in it, because we never repeat a request, and after k distinct requests you're at the end of a phase. So we've really simplified the request sequence down.

So now we're looking at a series of phases. Let S_i be the set of pages in memory at the *start* of phase i. This means we have a set of pages S_{i+1} at the end of phase i (which is also the start of phase $i+1$).

Given this set of pages, we're going to distinguish two types of requests.

Definition 25.4. We say a request in phase i is clean if it's not in S_i, and stale if it is in S_i.

From an intuitive perspective, these are two rather different kinds of request. If a page was present at the beginning of the phase, then you kind of assume a good algorithm should still have it around if it's going to be requested. So stale pages are ones you'd like to hang onto and hopefully won't miss. A clean request hasn't been seen for ages, because it wasn't requested at all in the previous phase. So it's kind of reasonable you would miss on a clean request — the only way to not miss on it would be to hang onto it for the whole previous phase, which would have no point because it's not used there.

Our intuition is going to be that for clean pages, the marking algorithm misses, but basically OPT does too. Meanwhile, for stale requests, we'll show that the marking algorithm is *unlikely* to miss. Putting those together will give us a good competitive ratio.

We have to analyze both the online and offline costs for this sequence. Let's look at the online cost first, and start by talking about stale requests. To set some context, let's suppose that at the point we're looking at a particular stale request, we have already seen s stale and c clean requests in the phase. (We need these numbers because they'll affect our analysis.)

If there have been s stale and c clean requests during the phase, this means there are $s+c$ marked pages in the memory, namely the things that have been requested. And we know that s of them are from S_i. We started out with S_i at the beginning of the phase, and there were s stale requests; those are still in memory. They might have been evicted before, but at least at this moment, they're back in memory.

Of the remaining pages, there's $k-s$ pages of S_i which are not certain to be in memory. And in fact, some were evicted — c of them had to be evicted to make room for the the clean requests. So the situation is that there's $k-s$ pages that might have been evicted, and c of them have to have been evicted.

So by symmetry, each of these pages was evicted with probability $\frac{c}{k-s}$ — our algorithm evicts random pages, it doesn't differentiate between them at all. So any two pages that are not themselves in memory due to having been requested are equally likely to have been evicted. And the total number of eviction has to be c and there are $k-s$ candidates, so each was evicted with probability $\frac{c}{k-s}$. Then this is our probability of a miss for the next stale request.

So we have our first bound on the expected cost of a stale request — the probability we miss is the probability the page is not present, which we just showed is $\frac{c}{k-s}$.

Now we can talk about the cost of a phase — the phase has k request. Let's suppose c_i of them are clean; since there are k requests, that means $k - c_i$ of them are stale. Now, the marking algorithm is not going to have any of the clean requests (they weren't asked for in the previous phase); so we pay c_i for the clean requests.

Now looking at the stale requests, when a stale request happens, there's some number c of clean requests that has already happened, and we know it's at most c_i. What we've just analyzed is that the sth stale request costs $\frac{c}{k-s}$ (where c is at the time of that stale request). And c might be smaller — if the stale requests come early, we might not have had the clean requests yet. But the worst case is if all the clean requests come first; so this is at most $\frac{c_i}{k-s}$. That means overall, we're paying at most

$$\frac{c_i}{k} + \frac{c_i}{k-1} + \cdots + \frac{c_i}{k-(k-c_i)}.$$

This is equal to

$$c_i \cdot \left(\frac{1}{k} + \frac{1}{k-1} + \cdots\right) \leq c_i \cdot H_k = O(c_i \log k)$$

(where H_k denotes the harmonic number).

This sort of starts to flesh out the intuition we made — we're paying for all the clean requests at a 1-for-1 cost, but we're only paying $\log k$ for every one of the stale requests.

But we do have to finish this up — we have to relate this to the cost of the offline algorithm. So let's look at the offline optimal algorithm OPT. It'd be nice to just say the offline algorithm has to pay for the clean requests — what's nice is we've bounded the cost of the clean requests against the number of clean requests, but we've alos bounded the cost of our stale requests against the number of clean requests, so we only have to worry about that number. So if we can just argue the offline algorithm pays for all the clean requests, then we get our $O(\log k)$ competitive result.

It'd be nice to say the offline algorithm has to be paying for them because they haven't come up since a long time, but that's not true — the offline algorithm may have hung onto them for a super long time. So we have to be careful about how we analyze this — we can't just count clean requests. So we'll use a potential function to *amortize* our offline algorithm — we'll say that in an amortized sense it's paying for the clean requests, even if it's not doing so in a real sense. For this, we'll define φ_i as the difference (in count) between the marking algorithm $\mathcal{M}$ and OPT at the start of phase i — in other words, this is k minus the number of pages that they have in common (in the cache).

With this potential function, we can amortized to argue that clean requests cost the optimum algorithm misses. We know we get c_i clean requests in phase i. These are *not* in $\mathcal{M}$'s memory at the start (i.e., they're not in S_i). That means at least $c_i - \varphi_i$ of these requests are not in OPT's memory at this time.

Now, at the end of the round, $\mathcal{M}$ has the k most recent requests. That means OPT is missing φ_{i+1} of them — because φ_{i+1} is the difference. And if they're missing from OPT at the end of the phase, what does that mean — how did they end up missing? Well, that means OPT evicted them. So we know there's a bunch of pages that were not in OPT's memory at the beginning of the phase, which then got requested (so OPT missed those). We also know OPT had a bunch of misses that caused evictions, by looking at the end of the phase.

This means OPT had at least

$$\max\{c_i - \varphi_i, \varphi_{i+1}\} \geq \frac{c_i - \varphi_i + \varphi_{i+1}}{2}$$

misses on this round. Why is this useful? If we sum this over phases, then the total misses by OPT over all the phases is at least

$$\sum_i \frac{c_i - \varphi_i + \varphi_{i+1}}{2} = \frac{1}{2}\sum c_i.$$

So in an amortized sense, the offline algorithm is missing on all the clean requests (if it hangs onto them to not literally miss on them, it pays for that by missing on something else).

So we've seen that whatever the sequence is, after we divide it into phases, the offline algorithm misses at least $\frac{1}{2}$ on each clean requests. Meanwhile our algorithm, in expectation, misses only $O(\log k)$ times for every clean request. Putting those together, we find that this randomized algorithm is $O(\log k)$-competitive on every sequence.

This is not just a small improvement — we went from k to $\log k$, which is an exponential improvement, just by introducing a little randomness.

§25.4.1 Can we do better?

There's the obvious question: can you do better? It turns out the answer is no — this is the best possible bound for a randomized algorithm against an oblivious adversary. But proving that takes some insights, which we might cover shortly. (There's two things we can do today — either that lower bound (if there are any complexity theorists here), or a generalization of the paging problem to k-server.)

§26 November 15, 2024

§26.1 Review

We got through deterministic and randomized paging on Wednesday. We've seen now that we can deterministically get a k-competitive algorithm for paging, and this is completely gith. We also saw that if you randomized, you could be $O(\log k)$-competitive, which is far better than k.

The natural next question is, can you do better? Is this the best possible for a randomized online algorithm? This really asks, how do you prove a lower bound on the competitiveness of a randomized algorithm? This seems really hard because there's a much larger space of randomized than deterministic algorithms — how can you put any limits on them at all? You can think of a randomized algorithm as a distribution over deterministic ones — you can write down all the possible sequences of random choices it might take, and every possible sequence gives you a deterministic one. So of course the space of randomized algorithms is vastly large than that of deterministic ones.

So what do we do? Andy Yao developed what's known as Yao's mini-max principle lower-bounding randomized k-competitive ratios. It uses the perspective that we just described. It recognizes that a randomized algorithm is a probability distribution over deterministic algorithms. The reason for this is that every randomized algorithm makes a series of random choices, or uses a series of random numbers. And each such series or sequence defines a specific deterministic behavior for the algorithm — in other words, it's a deterministic algorithm once you've fixed the sequence of choices.

§26.2 Game theory

And so what Yao did is, he tied this to game theory, which is another place it's fundamental to be making random choices over a space of possible strategies for playing a game. Game theory was studied by John Von Neumann, who did some of his work with Morgan Stern; they tried to formalize the notion of games and developing strategies to play them.

One of their simplest models of games is *zero-sum 2-player games*. You have two players, who play against each other; and based on the outcome, one player pays the other some amount (the net change in total money is 0, which is why it's called zero-sum).

We'll think about *extensive form*, where we have a list of strategies for each player. A strategy is interpreted very broadly — for every possible state of the game, it records a next move by that player. So for ever ypossible chess position when it's my turn to know, what's my next move going to be? You can imagine writing that down in a book; that's an extensive form representation for how to play chess. Of course, there are many possible books (you have a combinatorial product of what move you'd make in every single state of the chessboard).

We can represent two players playing against each other using a *payoff matrix*, where we have rows for player 1 strategies and columns for player 2 strategies. The (i, j)th entry is the payoff from 1 to 2 if 1 plays strategy i and 2 plays strategy j. If you know what strategy each player's going to play, then you can run the game and see the outcome and see what payoff happens. If it's positive player 1 is paying player 2; if it's negative player 2 is paying player 1.

What interesting things can we say about games in this level of generality? Well, let's look at a simple game, like rock–paper–scissors. This has the payoff matrix

$$\begin{bmatrix} 0 & 1 & -1 \\ -1 & 0 & 1 \\ 1 & -1 & 0 \end{bmatrix}$$

(with indices rock, paper, and scissors in that order).

§26.3 Optimal strategies

We're interested in figuring out the optimum strategy; which row is the optimum strategy for player 1? What's the best row to use? The answer is, there isn't one. Why isn't there a best row? The payoff doesn't only depend on my row, but also the second player's column. And a given row I pick is good for some columns, but bad for others. In fact, there is no row I can pick that's good for all the columns. Going further, in fact, if I announce my strategy — if I make public which strategy I'm going to play — then the second player has a tremendous advantage. Whatever row I pick, the second player can pick a column that's really bad for me.

As with competitive analysis, Von Neumann was thinking from a pessimistic perspective — we assume the opposing player is infinitely rational and so on so will always do the best strategy against me. So it seems I'm sort of doomed — there's no row that will let me do well.

But the way to fix this is through randomization.

Pure strategies — i.e., single rows — don't do well. If I announce my row strategy, then the column player picks a great strategy against me. If they announce that I'll change my row strategy, but then they'll change their column strategy; and you'll get this continuous mutation back-and-forth as people share information. That formally means there's no equilibrium — we never settle down with each of us having selected a strategy so we can start playing a game.

But this can all be fixed by using a *mixed* strategy.

Definition 26.1. A mixed strategy is a probability distribution over pure strategies.

We can represent that probability distribution as a vector for player 1 summing to 1, with all nonnegative entries. And for player 2, we can do the same thing. We'll call these vectors p and q, respectively.

Once we have these vectors p and q representing probability distributions over strategies, now we're randomizing; there's no longer a single outcome for the game (it depends on the randomization), but we can talk about the *expected* payoff — we have

$$\mathbb{E}[\text{payoff}] = \sum_{i,j} p_i q_j M_{ij} = pMq.$$

So there's a nice simple formula for the payoff, given these mixed strategies.

What Von Neumann proved is that:

> **Theorem 26.2** (Von Neumann)
> *Every* zero-sum 2-player game has optimum equilibrium strategies — i.e., a pair (p, q) such that even knowing the strategy q by the second player, player 1 still prefers their strategy p over anything else; and simultaneously, even knowing p, player 2 still prefers q over anything else.

Going back to our rock-paper-scissors example, the pure strategies of putting probability 1 on a row are not equilibrium strategies (whatever the two players have chosen, one is motivated to change their strategy when knowing about the other player's plan). But if I announce I'm going to choose equally etween the three pure strategies, and the second player announces the same thing, then neither of us has any incentive to change our strategy. If I know the second player will pick randomly among the three strategies, then I might as well do that too — I can't improve on my payoff with any other strategy.

This is actually the only equilibrium — if there's any imbalance where one column is preferred, I will choose a strategy that counters that (a best-response strategy plays against the most likely of these three strategies, and earns a better payoff).

And you can actually prove this using LP duality — this was an optional problem on a pset.

So now we're going to use this.

Another interesting fact, which we used in this analysis is:

> **Fact 26.3 —** If one player announces a randomized strategy, then the second player has a *deterministic* (i.e., pure) best response.

Proof. Suppose I announce my strategy p. We claim the second player has a pure strategy that's optimal. How do I figure that out? We said that the expected payoff is pMq. But I've just declared that I'm fixing p, so I can go ahead and compute pM; what comes out of that is a vector. And now for the second player's strategy q, their payoff for any q is a dot product with this fixed vector — it's an expectation, which is just a weighted sum of the entries in this vector.

So what is the optimal q? You put all the mass of q on the maximum coordinate of this vector pM. We're putting all our mass (i.e., probability 1) on a single coordinate, so that's a pure strategy. □

The same holds in the opposite direction — if q is announced in advance, then there's an optimum pure strategy for the first player (which goes for the maximum-value column of Mq).

We proved this key theorem in an optional homework (which we're welcome to go back to if we haven't done it already).

§26.4 Yao's minimax principle

Now we can go back to randomized online algorithms.

We're going to look at randomized online algorithm design as a game. There's an adversary out there who's picking input sequences — their strategies are possible input sequences. And our strategies are possible algorithms for input sequences. In a sense, the payoff matrix is the cost of running that algorithm on that input sequence. So now we can see the adversary is trying to make my cost as high as possible, and I'm trying to make my cost as low as possible. So we can represent this in a zero-sum way and turn this into a

two-player zero-sum game. Once we've done this, we see randomization is necessary to get an equilibrium, and the best strategies will be randomized.

We already saw an instance of this with randomized competitive algorithms. But in this framing, we can also think about randomized strategies for the adversary — instead of the adversary picking just one sequence (where we'd have a good algorithm for just that sequence), we can think of them as picking a probability distribution over input sequences.

But at equilibrium, only one of those has to be random; the other can be deterministic. And we can leverage that.

Theorem 26.4 (Yao's minimax principle)

If there exists a distribution over input sequences such that no deterministic algorithm has expected competitiveness k, then there is no *randomized* algorithm with expected competitiveness k.

We should tighten up our definitions a bit:

Definition 26.5. We say an algorithm is k-competitive versus a *distribution* over σ if

$$\mathbb{E}_\sigma[c_{\mathcal{A}}(\sigma)] \leq k\mathbb{E}_\sigma[c_{\mathsf{OPT}}(\sigma)].$$

So we just put expectations around everything to understand what's happening when we talk about randomized algorithms. What's important is that OPT knows σ. But also, both of us know the distribution. So the order of activities here is we consider a particular distribution over inputs; and I'm allowed to design my algorithm to be as good as I can for that distribution of inputs. But I still can't do as well as the adversary, because they don't just know the distribution of inputs, but also which *specific* input it's running on.

Proof. All we do is translate this into a payoff matrix — I'm picking a distribution over algorithms, and the adversary is picking a distribution over input sequences. The payoff we write down for a given algorithm and input sequence is just

$$c_{\mathcal{A}}(\sigma) - kc_{\mathsf{OPT}}(\sigma).$$

Now we're going to talk about the expectation of this quantity. If I have a k-competitive algorithm, that means the expectation of this quantity is nonpositive — so

$$\mathbb{E}[c_{\mathcal{A}}(\sigma) - kc_{\mathsf{OPT}}(\sigma)] \leq 0$$

(by the definition for k-competitiveness).

Now let's do a proof by contradiction. Let's assume that in fact, there is a k-competitive randomized algorithm. What does that mean? Now we go back to the point that only one of the players has to randomize at equilibrium. So what's the equilibrium strategy for the two players in this game? It's a certain distribution over inputs, and a certain distribution over algorithms. What we said before is that the player who chooses second can be deterministic — so if I start by choosing a randomized algorithm (a probability distribution over algorithms), then the adversary's best response is a particular sequence on which I do poorly in expectation. But if we turn that the other way around, if the adversary goes first and picks a particular distribution over sequences, then my best response is a specific deterministic algorithm that'll do as well as any randomized one given that particular distribution over sequences.

So if there is a k-competitive randomized algorithm, by our standard definition, that means

$$\mathbb{E}[c_{\mathcal{A}}(\sigma) - kc_{\mathsf{OPT}}(\sigma)] \leq 0,$$

and this is true *even if* the adversary chooses their best *randomized* strategy — because there's no randomized strategy that does better than this.

But now if I instead consider the adversary's best randomized strategy, given the best adversarial σ, there exists a deterministic algorithm $\mathcal{A}$ that achieves the same expected payoff, which by assumption is nonpositive.

In the general case, this is an expectation over both algorithms and input sequences. But if I'm choosing my algorithm after the distribution is known, then my algorithm will be a deterministic one.

So we've proven the contrapositive — if there is a randomized algorithm with expected competitiveness k, then there's no distribution the adversary can offer that will make for a positive payoff. This means even if the adversary chooses a randomized strategy, I can always figure out a deterministic strategy that always achieves a nonpositive payoff.

So if there is a randomized algorithm with expected competitiveness k, then for any distribution over sequences, there is a deterministic algorithm achieving expected competitiveness k. (Note that this algorithm will depend on the distribution.) □

Remark 26.6. Note that it's easier to be competitive against a distribution than against an unencumbered adversary, because you have information about that distribution — for example, you can just not worry about the possible sequences that aren't in the distribution.

§26.5 A lower bound for randomized paging

Now let's use this. We want to prove that there is no better randomized algorithm; so we need to give a distribution over sequences such that no deterministic algorithm, knowing the distribution, beats our $\log k$ bound.

The distribution over sequences is simply going to involve a set of $k+1$ pages, and the request sequence is uniform random. Really if you think about this, this is a very depressing situation for a computer — it has room for k pages, but the algorithm has $k+1$ and there's no useful information (you know the next request is completely random, so you can't imagine doing any useful planning).

But our limitation is that we don't know the future. The adversary gets to pick the algorithm after the entire sequence has been generated, so it can do better.

The point is that any deterministic (or randomized) algorithm has a fault probability of $\frac{1}{k+1}$ on each request — just the probability that the page you don't have is asked for. So in n steps, we expect $\frac{n}{k+1}$ faults.

How can we take advantage of knowing the future? Let's think about a prophetic algorithm. If you know the entire future of requests, and you get a miss, what's the optimum page to evict? What would be the right greedy strategy for eviction? The optimum algorithm is called *farthest in future* (FIF); you can prove that this is optimal for a given sequence. Basically, if you evicted any page that's not farthest in the future, then you're going to get a miss earlier. And at that point, you can switch back to evicting the other page that's farther in the future — you can show by a simple exchange argument that evicting something that's closer is only worse for you (or no better).

So to understand the performance, we just need to understand, when I have a miss and evict a page farthest in the future, how far in the future is that? The randomized algorithm gets a fault every $k+1$ steps or so. How many faults happen between steps for the FIF algorithm? Well, the question is, how far in the future is the next fault?

How would we model this? I'm evicting a page. Let's imagine I haven't yet looked at what's farthest in the future; and now that I have a miss, I'll figure out which page that is and how far it is. So I'm now going to start generating a series of requests, which are all uniform random; and I'm going to keep generating those requests until I find the page that's farthest in future. What does that mean — how long am I going to be generating those requests?

Another way to look at this is that as long as there are some pages I haven't seen yet, I haven't yet gotten to the page farthest in the future — the FIF is determined by when I encounter the last page. So at each step I'm generating a random page, and I'm waiting until I see the very last of those $k+1$ pages — I'm waiting until I see all of those $k+1$ pages.

How long does it take me to see all the $k+1$ pages, if at each step I get a random page?

Suppose that instead of talking about pages, I talked about baseball cards — I'm getting random baseball cards, and I want to know how long until I see all the baseball cards. This is the *coupon collector* problem!

So on a miss, there are no more misses until the randomized request sequence requests every page. And how long does that take? Well, it's the expected time to get a first new page, plus the expected time to get a second new page, and so on, up to the expected time to request the $(k+1)$st new page. Now, we get the first new page immediately (1 request). When I have one page already seen and I start generating random pages, the probability each request is a different page than that one is $\frac{k}{k+1}$, because there are $k+1$ distinct pages. So the expected time until I see that second page is $\frac{k+1}{k}$.

And for the third page it's $\frac{k+1}{k-1}$, and then $\frac{k+1}{k-2}$, and so on; in the end, there's only one page left which I want to see, which means it'll take me $\frac{k+1}{1}$ steps to get that page (in expectation). So it's $k+1$ times a harmonic sum, which sums up to $(k+1)\log(k+1)$ (in expectation).

So the online algorithm faults every k steps or so, but the prophetic algorithm only faults every $k \log k$ steps or so. This means the prophetic algorithm faults $\frac{1}{\log(k+1)}$ times as often as the online one.

So we have a specific distribution over input sequences, and we just showed no deterministic online algorithm does better than $\log k$-competitive on that specific distribution. Going back to Yao's minimax principle, if there was an algorithm better than $\log(k+1)$, then there would be a deterministic algorithm that could beat $\log(k+1)$ against every distribution, in particular this one.

So Yao's principle is a beautiful way to take the complexity of randomized algorithms out of the algorithm — I need to consider a distribution over input sequences, but then I only need to consider deterministic algorithms against those input sequences.

> **Remark 26.7.** We skipped over some technical details. When we're talking about randomized algorithms, there's a question about the degrees of knowledge the adversary has (obliviousness and so on). The lower bound we proved here is assuming an oblivious adversary — it's aware of your distribution but has no knowledge of the choices your randomized algorithm is making. If the adversary knows about the choices, then all of this breaks and you have to analyze things differently.

Looking ahead a bit, next Wednesday is our last problem set. David wants to check whether there are any questions about the project. At this point, we've seen the project handout. We should email in the project proposals, and they'll try to give us responses quickly — it's a weird year where Thanksgiving is late, so we don't have that much time after.

§26.6 The k-server problem

We'll finish with some 'cultural awareness.' There's a famous problem in online algorithms that we should be familiar with, called the k-server problem. This was defined by Manasse et al in 1988. They basically generalized the paging problem to a crazy degree, and then asked about the competitive ratio of this generalized problem.

It's defined using a metric space with k servers moving among the points in the metric space. Each request is a point in the space. And to serve it, you have to move a server there. Of course, the flexibility is that you get to choose which server you move.

We're looking at an online version of this problem — you know the metric space in advance, but the requests come one at a time, and each time you get a request, you have to move a server to cover that point, without knowing what points are going to be requested in the future.

Student Question. *Where do the servers start initially?*

Answer. This is a fair question for any online algorithm. We didn't really talk about this with paging (perhaps you start with the cache empty). But in most online algorithms the initial state doesn't really matter — as the request sequence gets longer and longer, it turns into an additive difference. So if the cost of all the requests grows without bound, that additive thing at the beginning is negligible. We defined asymptotic k-competitiveness in the beginning for exactly this reason. People tend to be sloppy about where we start things because in the long run it doesn't matter.

The cost is the total distance moved.

(You could have any metric space — it might be a small number of finite points, or it could be $\mathbb{R}^3$, which is unbounded. In particular, we're not limiting ourselves to discrete metric spaces. Some of the algorithms only work for discrete metric spaces, but it's not fundamental to the definition of the problem.)

This is an elegant natural formulation of a problem. The most obvious story doesn't quite work — you could imagine a story about taxicabs where people order taxis, and you have to send taxis to pick them up. But this doesn't really quite capture what's going on — with taxis, when you pick someone up, you have to drop them off somewhere else. If you tried modelling this as another request, you might send a different car to the new location, which won't help.

Still, this does capture some natural problems. One natural example is repairmen — imagine something's broken and you have to send a repairman to fix it. There's a sense where you want each repairman to not have to travel too far.

And it does generalize paging. How can you translate a paging problem into a k-server problem? The servers represent what's in your cache; and the metric is the uniform metric, where moving anywhere is a constant cost (moving between any two points costs 1, because it represents evicting one page and putting the other one into your cache). But it also captures various kinds of weighted paging, where loading different pages costs different amounts — some might be in faster storage and some in slower storage, and you can take that into account in this more general model. Also back in the days of disk drives, this captured the use of multiheaded disks (disks had right heads that had to move between different tracks; moving a head to a different track took time, so you don't want to move them too far; your requests are tracks on disks and you're deciding which head to move to the track).

There is an offline solution, by reduction to min-cost flow. We're welcome to think about how to do that, but we're not doing graph optimization anymore, so we won't talk about it here. But it can be done.

Question 26.8. How can we approach this more general version of the problem?

§26.6.1 First attempt — greedy

What's the most obvious algorithm for k-server? It's the greedy algorithm — move the closest server. But here's a simple example where greedy doesn't work. Suppose we start with three points, with a server on two of them (2 and 3); and we repeatedly alternate requests between points 1 and 2 (where 1 and 2 are close to each other, and 3 is far away). Then the left server keeps going back and forth between them. Meanwhile, what *should* happen is you move the right server to the top location — if I'm going to have this incredibly long sequence of requests for those two points, then I should move this server over to the second, and then that sequence becomes free.

This might remind you of the ski rental problem. Moving this one server back and forth is like renting skis — it's how you want to start, but doing it forever gets crazy expensive, and at some point you should make the decision to move this extra server over. Greedy will never do that, so greedy is not competitive.

§26.6.2 Harmonic algorithm

It took a long time to figure out whether it's possible to be competitive at all — because you need an algorithm that sometimes moves a faraway server. Ski rental gives a godo intuition — you should think about faraway servers when the total cost you've spent is proportional to how far it'd have to move. Fiat et al (Fiat was also who broke the factor of 2 bound for PCMax online, and who developed the marking $\log k$ algorithm for paging) developed the *harmonic algorithm*, which applies the ski rental idea — you move a server with probability proportional to 1/distance. It's kind of randomized ski rental — you basically buy with probability $1/t$, so after about t timesteps, you will buy. Similarly here if you have something a great distance away, after that distance number of timesteps you'll consider moving that server.

They showed this has a competitive ratio of $O(k^k)$. The firsrt algorithm had a competitive ratio which looked like $k^{k^{k^{\cdots}}}$, so it was a big improvement to prove a finite ratio; but it's barely finite.

Then peopel developed a potential function called the *work function* that they thought would be a good hammer for cracking this, which tries to track how much it'd cost to shift from your current state to what the state would be if you ran the optimal algorithm all along. So it tries to measure the distance between you and OPT, and tries to minimize not just your cost but also incorporates the cost of shifting from you to OPT. This seemed a good idea, but no one could figure out how to use it, until in 2001 an algorithm based on this work function was proved to be $2k$-competitive.

So we have this massive generalization of paging, but we're able to get pretty close on the competitive ratio to paging. As far as David knows, this factor of 2 is still there; the work function algorithm is conjectured to be k-competitive, but this is not proven. (This algorithm is deterministic.)

Computing the work function takes exponential time, so this isn't polynomial-time; but it achieves a good competitive ratio. The question of computing the work function in polynomial time, or getting some other polynomial-time competitive algorithm, is still open.

§26.6.3 k-server on a line

The analysis of the work function algorithm is quite complex, and we're not going to see it. But we'll see some analysis of how you can do k-server on a *line*, which is an example of a metric space (now we're doing online algorithms on a line, which is kind of fun).

The example showing greedy is bad could be seen as being on a line; and we've seen the intuition that you kind of want to move the faraway server when the total cost of serving grows to be the distance of that faraway server. We're going to build on that idea.

Imagine I have a bunch of servers on a line, and I get a request. What can we say about the optimum strategy? Suppose that I *did* know the future (I know all future requests). Can we limit what might happen in the optimal offline algorithm?

Is it possible that the optimal offline algorithm would serve a request with the rightmost server (where we have SSSSSRSS)? This server is going to sstart here and start moving towards the request; at some point it'll cross the second-rightmost server. We claim this is pointless — if I'm goign to move this server here, and then cross; then instead of doing that, when this server arrived here, I could swap their identities and start moving the closer server instead. So eventually this closer server is going to reach the request. That tells me that without loss of generality, the optimal algorithm will serve the request using one of the servers that is closest to the request (there's one on each side).

Now, the optimum algorithm could also move some other servers around, but what's the point? If I actually do move the rightmost server to the second-rightmost, and then move the second-rightmost to the request, that didn't accomplish anything — I could pretend that it's here, but defer the actual moving of this server until later.

So the optimum algorithm will serve using one of the closest servers, and won't move anything else. This really narrows the complexity of what we need to think about — we know it'll move one of these two servers to the request, and nothing else is going to change.

So now that we've narrowed it down to just two possibilities, what should we do? We know that greedy doesn't work¿ But what can we consider instead? As with Vertex-Cover, if we know OPT is moving one of them, why don't we move both?

Now, how far should we move both? We know OPT is moving one of them, but it might be moving the closer one. So if we bring both to the request, we may be spending a tremendous amount more than OPT; so we shouldn't do that. But what would make sense is you take the minimum of the two distances, and move both that distance towards the request.

This is known as the *double coverage* algorithm.

Algorithm 26.9 (Double request)

On a request: if it's outside the span of the servers, then we move the closest server to it (e.g., if all the servers are on the left of the request, we take the rightmost server and move it).

Otherwise, we move the nearest servers on each side towards the request, by equal distances, until one reaches the request.

Student Question. *Why not just keep the further server where it is?*

Answer. If you actually use this in practice, moving both is very good for analysis, but in practice we'd leave the farther server where it is and just move a virtual copy to where we're imagining it to be for the analysis, and only actually move it when it's used to serve a request. But in the analysis we'll see why we want to move both for the analysis.

On Monday David will show us the analysis of the double coverage algorithm. Then we'll be done with online algorithms, and we'll talk a bit about computational geometry and maybe a bit about external memory or parallel algorithms. After Thanksgiving there will be optional lectures if peopple want.

§27 November 18, 2024

§27.1 The k-server problem

Last time we introduced the k-server problem. This generalizes paging, as well as weighted paging (where different pages have different costs for being brought in), and models a variety of real-world problems. This was introduced in 1985, and there was lots of progress in the 2000s.

We're looking at the special case of k-server on a line. This is a metric space with infinitely many points, but that doesn't matter for this algorithm (there are algorithms whose competitive ratio depends on the number of points, but this one doesn't care; its competitive ratio only depends on the number of servers).

§27.1.1 The double coverage algorithm

Last time, we applied this argument that given a starting state, without loss of generality you can serve a request with the closest server on one side or the other — crossing servers is never necessary. (These are

bosons, not fermions. You can't tell them apart, so you might as well use the closest one.)

Our idea, kind of lifted from vertex cover, was to say that since *one* of these is the 'right' server to use, we should use both! So we're going to move both of them towards the request, by a distance equal to the smaller of the two distances (so the closer server will reach the request, and the other will get closer but won't reach).

Actually moving the second server is pointless (it's not going to serve the request); but the reason we're doing this is because it simplifies the analysis. If you actually implement this algorithm and use it, then you shouldn't bother moving the server. Instead, you should keep track of a virtual location of the server (where the algorithm wanted to move it to) and keep running the algorithm with these virtual locations, only moving the server that actually serves the request. The virtual algorithm will upper-bound your total serving cost, but you might do much better by not unnecessarily moving things (procrastination/laziness).

As a subtle point (which someone asked after class): if we have this argument that we're moving one of the correct servers, does this immediately let us conclude that we have a 2-competitive algorithm? That would be great, but there's a problem. The heuristic argument we made said that OPT is going to have to move one of these, but that's assuming that OPT is starting in the same state that we are — that argument that OPT moves at least half as much as we do only applies if OPT starts in the same state that we do. It's possible OPT already has a server on that state, in which case it doesn't move anything at all. We're doing at most a factor of 2 from what we have to do, but this could be much worse than what OPT has to do. And as we execute this procedure, we may be making our state more and more unlike OPT, which means our cost will be nothing like OPT's.

So we have to amortize the work we do over requests, recognizing that OPT may be in a different state. Last time we mentioned the work function, which was introduced because of this recognition that you have to worry about getting too different from OPT's state — then you may pay a large cost while OPT pays a low one. The work function measures not just the work you have to do to serve a request, but also how different you are from OPT. You can't compute that, but you can use that for analysis; and you want to keep that small, to make sure OPT is paying something like what you are paying.

§27.1.2 The distance to OPT

Now let's look at our particular algorithm; how are we going to analyze it? Given what we just said, we'll explicitly model the difference between us and OPT — in particular, how much work we'd have to do to make our state the same as OPT. We'll explicitly keep track of that as part of our analysis.

We have a task — here's where the OPT servers are, and here's where ours are. What problem do we need to solve to find the min-cost way to make our servers the same as OPT's? This is min-cost bipartite matching — we need to decide which of our servers is going to move to the location of which of OPT's servers. So we're pairing each of our servers with one of OPT's, and the cost is the distance between each pair. So we're solving a min-cost bipartite matching problem to understand the gap between our servers and OPT's.

So we're going to keep track of the distance m between our algorithm (the double coverage algorithm, which we call DC) and OPT, which is just the min-cost matching between DC and OPT under the distance-on-the-line costs.

In terms of computing that optimal bipartite matching, if you think about how to find it, you don't need to run a complicated algorithm. The point is there's some leftmost point, either in OPT or DC. Which point should get matched to that? We claim it's the leftmost point in the other algorithm — because if we match a different point to that one, then when we're moving that point to its destination, it's going to cross another server (specifically, the leftmost one). And there's no point to doing that crossing — at that point, we can relabel the two servers.

So in fact, we match the leftmost DC point to the leftmost OPT point, and so on — we just take them in their order, and that's the matching.

This is going to show up in our analysis.

§27.1.3 A potential function for amortization

We'll also need another quantity; it's not clear why yet, but it makes the analysis work. We'll also talk about a sort of *total distance* — this is also a distance between all pairs of servers in the double coverage algorithm, in other words $\sum_{i<j} \text{dist}(s_i, s_j)$. This makes no reference to OPT at all; it's just the distance between points in our algorithm.

We'll use these quantities in an amortization. In an individual move our algorithm might spend much more than OPT, but we'll use amortization to show on average it doesn't. We'll do this by using a potential function, which we define as

$$\varphi = k \cdot \text{matching distance} + \sum_{i<j} \text{dist}(s_i, s_j)$$

(where k is the number of servers).

We'll also model this as follows (for the analysis) — on a request, we'll have OPT move first, and then DC moves. (That's just how we'll do the accounting.)

We'll also assume that all the servers start at 0. (We need some starting configuration. This doesn't matter asymptotically — it adds a fixed cost to things — but this one is convenient for the analysis.)

§27.1.4 The analysis

We'll show two claims:

> **Claim 27.1 —** When OPT moves a total distance of d, this increases the potential by at most kd.

> **Claim 27.2 —** When DC moves d, this decreases the potential by at least d.

First, we claim that if we can prove these two things, then we've proven that our algorithm is k-competitive. Why? The potential function can never drop below 0. When our algorithm moves, we can think of this as spending potential to pay for that move (I'm taking money out of the bank to pay for my move). If I move d, I'm taking more than d money out of the bank. Where did that money come from? Well, the initial potential is 0; so in order for there to be money in the bank, OPT has to have deposited money into the bank.

So if I move a total distance of d over my algorithm, there has to be a total of d money deposited in the bank (at minimum, since at minimum I pull this out). And this can only get deposited by having OPT move; and in order to deposit a total of d, OPT has to move by at least d/k to deposit enough to pay for my movement.

So let's prove these two steps. Let's first consider what happens when OPT moves by some distance d. How much can this change the potential function? There's one server moving (it never makes sense for OPT to move two servers at the same time — you can save the second one's move until it's needed). What can this do to the potential? We can ignore the second term, which only plays a role in DC (moving an OPT server doesn't affect it). So the only change in potential happens to the matching. OPT is moving a server, which is matched to something. It might get closer to its match, which means the potential decreases. The worst that can happen is that it gets farther away; if it moves a distance d, that at worst increases the distance and therefore potential by d.

We say *at worst* because you could cross another server, which would actually change what the minimum cost matching is — maybe now it's moving towards its new match, and the potential is going down again. So it's not exact; but at worst, OPT's server moves away from its match and increases the match cost by d.

So that was the easy part. Now let's think about what happens when DC moves. For this, there's actually two cases we need to consider. We briefly talked about and discarded the edge case where the request is outside the range of the interval spanned by our servers. Our servers have spread out from 0, but the line is infinite, so the request could come from beyond the farthest distance any of our servers have moved. So let's take care of that case first.

Case 1 (We have a request outside the interval spanned by our servers). Now in that case, there's only one side of that request, so we're moving one server, namely the closest server (the one that's outermost to the request). What does that do to our potential?

Now we have to remember the potential comes in two pieces — a km term and a sum of distances term. The server we're moving is the rightmost DC server, and it's matched to the rightmost OPT server. And we set things up so that OPT moves first, which means that server is already on that request. So the server we're moving is matched to the server at that request, which means we are decreasing our cost by the distance we moved — we're decreasing that matching cost from d to 0.

What can we say about the change in the summation cost? This server is moving away from all the others, so its distance from all the other servers is increasing, by the distance that we moved. So here we get an increase of $(k-1)d$.

And that means if we add them up, we multiplied the matching cost by k, and the reason for that is now we have

$$\Delta\varphi = -kd + (k-1)d = -d,$$

as we asserted. That's why we need the factor of k — to make that argument work out.

Case 2 (DC moves two servers inside the spanned interval). Notice that we're talking about a cost of d, which means that each server moves by $\frac{d}{2}$ (because we're moving both of them). Our analysis isn't in terms of the distance of one server, but the total distance we move to serve a request.

What does this do to the matching cost? The argument is that one of these two servers is moving *towards* its match. Why? If both are moving away from their match, then the left server wants to match to its left, and the right one to its right. But these are adjacent in DC, so they should match to adjacent covers in OPT. And there's a server in between them, so something has to match to this! If it's on the left, then the left server should match to something on the right; and likewise in the oppostie direction.

So one of the servers is moving towards its match. That server *decreases* the cost of this matching by $\frac{d}{2}$. The other server we don't know — it might increase the cost by $\frac{d}{2}$. But what that means is there's no net increase — the decrease by one server is at least as large as the increase by the other. So m does not increase as a result of this double move.

This is where our double movement trick is working — we're saying OPT basically would want to move one of these. We don't know which, so by moving one of them we get the benefit of what OPT would do. It's possibly cancelled out by what OPT wouldn't do, but it's only cancelled out; it's not made worse.

And what's the change in total distance? It decreases by d. We only care about the sum of distances to the two points that are moving. For all the other points, when we move both of these points, their average distance from every other point is unchanged. So the contribution to the sum from these two points vs. any other point is 0. The only change that isn't cancelled is the distance between these two points, and that decreases by d.

So our distance changes cancel between the two points, except for the distance between the two moving points. And of course, that change in distance is exactly $-d$ — they get a distance d closer to each other. That completes the case analysis — in the case of a move outside the interval we proved a change of $-d$ in the potential (with the appropriate scaling factor); in the second, we also proved a change of $-d$, which comes entirely from the sum-of-distances term.

So with these two cases complete, we've proven the second critical claim,that potential decreases by d when DC moves. That completes our assertion, which gives the k-competitive ratio.

Student Question. *Are the two servers moving adjacent?*

Answer. Yes, because it's the closest one on the left of the request and the closest one on on the right — so there can't be anything between.

Student Question. *WHy do we need the coefficient of k in φ?*

Answer. This comes from the first case. We need to end up with $-d$. If we make the coefficient on m smaller, then we won't be able to cancel out the positive increase in $\sum \text{dist}(s_i, s_j)$, so we won't be able to get the decrease in potential that we need.

Prof. Karger doesn't have a good insight as to how they came up with this potential function. But the matching is a very natural potential function to work with, related to the work function (the work function was introduced in the original paper on k-server — they already thought it'd be interesting, but couldn't prove it).

§27.1.5 Improvements

As always, there's the question, can we do better? We can't do better than k-competitive for k-server because of the paging lower bound, at least if we're talking about deterministically. It might not be optimal for the line — the line is not a generalization of paging — and Prof. Karger hasn't thought about lower bounds for the line.

But as mentioned last time, two people (KP) proved a $2k$ bound for the *general* k-server problem, using the work function. No one knows how to compute the work function efficiently, so this isn't polynomial-time, but it is $2k$-competitive. As far as Prof. Karger knows, this gap between k and $2k$ remains, and we're not sure where the tight bound is.

Later, Bartal introduced a very interesting technique called *metric embeddings*. He gave a *randomized* competitive ratio of $O(\log^3 n \cdot \log^2 k)$. This is exponentially better than the k you could achieve deterministically, but brings in a dependence on the number of points in the metric space; so it does not work for the line, but does work for the discrete line. (This involved *metric embeddings*.)

That's it for online algorithms.

§27.2 Computational geometry

Our next topic is computational geometry. This means what it says — algorithmic problems involving geometry, i.e., problems involving points, lines, half-spaces, curves (sometimes), and so on. From that perspective, we've already done some computational geometry when looking at linear programming, which is a very geometric problem. But classically, computational geometry looked at *low dimensions*. Lots of early computational geometry was driven by computer graphics, which involves 2D or 3D or sometimes 4D. It turns out that in low dimensions there are techniques you can apply that are very powerful, but don't work well in high dimension. That's what we'll demonstrate.

More recently there has been work in high-dimensional computational geometry. For example, one major topic is near neighbor searching, which is very important in machine learning (where you have vector embeddings and want to find the closest vector). But we're not going to look at that; we'll focus on the low-dimensional version of the problem.

In the low-dimensional version, a key idea is *dimension reduction*. If you have a problem in 2D and you can find some technique, that perhaps at an algorithmic cost, reduces it to a 1D problem — 1D problems are easy (e.g., on lists or something). So even if it's costly to do the reduction, it turns it into an easy problem

you can solve. And then you can reduce 3D to 2D. Each of these reduction steps might be costly, but since we're only in a small constant dimension, those costs don't grow too much.

We've already seen an example much earlier, when we talked about a point-location data structure using persistent trees. We had a bunch of line segments dividing space into polygons; you get a query, and want to know which polygon the query point is in. We solved this by imagining we ran a *sweep line* across the space, and keeping track of how the binary search tree changed as we moved through the geometry. So we had a 2D problem, and wanted to instead think of it as a 1D problem we could solve with a binary search tree, that evolved over time.

We'll use a very similar idea for a related problem, the first one we'll see.

§27.3 Orthogonal range searching

We'll focus on the 2D version, though this generalizes to arbitrary dimension.

Example 27.3

We're given a set of points in space, and we receive a query, which is a *orthogonal range*, meaning a rectangle (a vertical range interesected with a horizontal range). We want to ask about the points in the rectangle. There are many things that could be asked:

- We mifght want to count these points.
- As a special case, we might want to know, is the rectangle empty?
- We might want to enumerate all the points in the rectangle.

(This is a data structure model where we have our points, but don't have a query until it's asked.)

We'll apply the same technique to all these problems. But one is clearly harder than the others, and is going to have limits on its time bound that the others don't — enumeration is going to necessarily take time at least the number of points in the rectangle. However, an empty query is just Boolean and counting just returns an integer, so there are no fundamental lower bounds to those.

§27.3.1 The 1D version

Whenever you do computational geometry, you want to start building your intuition with a simpler version. Let's consider the 1D version. Here we have points on a line, and a query interval. How do we solve this, and what lower bounds might we be able to derive? A binary search tree is one answer. This is actually oversolving the problem — a BST is good if you have insertions and deletions, but we're just thinking about a fixed set of points.

We can just sort the points — now when we get a query interval, we can just ask where in the list its left and right points are. Then I can immediately tell you whether the interval is empty, I can enumerate easily, and I can count (if I have an index for different points).

So this is going to take $O(\log n)$ time to do the binary search within the interval. Or I can use a binary search tree if I want to be able to insert and remove points, and I'll achieve the same time bounds. (It takes a bit more work to count items, but this is 6.046 stuff, and you can see how to do it.)

Note that this is actually tight — I can use binary search to find the endpoints of the interval, but that's actually the best I can do. If we assume comparison-based algorithms, there's a lower bound that you have to ask $\log n$ questions to figure out where you land.

So 1D is easy — we have tight optimal algorithms. Now let's try to generalize this to 2D.

§27.3.2 Generalizing to 2D

The first obvious generalization would be to put a binary search tree on x, and another binary search tree on y — our query is an x-interval intersected with a y-interval, so we can make a BST on x (telling us what goes on in the x-direction) and another on y. How will this do?

As a bad example of what could happen, suppose we want to do an enumeration query, and suppose we have a bunch of points in a top-left square and bottom-right square, and we get asked about a top-right square. If we look at the x-coordinates we get a nice collection of points lying in the vertical direction, and y-coordinates gives us a nice collection of points in the horizontal direction, but that doesn't tell us about what's in the rectangle. We could imagine enumerating all the points in the slabs and intersecting them, but that would take a huge amount of work.

So we can't think about solving the two directions independently. But can we refine this? Suppose I search in the x-coordinates, and I discover that a certain thing is the slab of interest for our search. We now need to find the points that are in the proper y-interval. What's a good data structure for doing that, if we're focusing on this particular slab? We can just do the binary search thing on y in that slab. This is the dimension reduction idea — once I've done the x-coordinate search, I really only have a 1-dimensional problem left to solve. So I can take these points in the slab and put them into a BST, and now it only takes us $\log n$ time to find all the points in our rectangle.

So just putting one BST on x and another on y fails. But given a particular slab, I can put a BST on the points in the slab, and answer questions quickly — I can answer emptiness, count, and enumerate. And now the runtimes are good, because it's just the 1-dimensional problem.

So that's great — we seem to have a solution. It only takes $\log n$ time to figure out the left and right sides of the slab, and if we've built the slab data structures in advance, it only takes $\log n$ time to do the vertical search. So that's $\log n$ time horizontal, and $\log n$ time vertical. So this is $O(\log n)$, assuming that we have these slab data structures.

So, how much work is involved in creating these slab data structures? Each one is just a BST, so that's not a big deal. But how many slabs do I have to worry about? You might say infinitely many — it's the real line, so the x-coordinates could be any real numbers. But we only need to worry about n^2 slabs — if two slabs have the same points in them, it doesn't actually matter what the coordinates of the boundaries are. And the points are ordered on x, and all that matters for defining the slab is the first and last point in the x-interval.

So there are only $\binom{n}{2}$ meaningfully distinct slabs. So we've got something — this implies that it takes $\binom{n}{2} \cdot n \log n$ (where $n \log n$ is the time it takes to build a BST on n points in one slab) time to build the structure.

That's a bit disappointing — it takes a long time to build the data structure — but query time is great. And this can be generalized to arbitrary dimension — we can do this trick over and over again for the 2D and 4D problem and so on. The query time will still be $O(\log n)$, though the polynomial for construction is not going to look so good.

§27.3.3 Improving construction time

What we're going to do is our x-coordinate search tree looks like some BST (we could even just use an interval if we're not planning to add and remove points, but we'll draw it out as a BST because it's easier to think about). We were talking about putting these slabs everywhere. But let's imagine what happens when we search our x-interval. We'll assume the points are all at the leaves. Then we're worried about points in a certain interval of leaves. We don't want to make slabs for every interval, but there's a natural set of slabs we can work with — we can make a slab associated with each *subtree* in this binary tree.

So it's better to take the x search tree, and make a slab for each subtree in it. The key insight is that every interval is a combination of disjoint subtrees, so we can solve the interval by solving and combining the subtree slabs.

And you can see which subtrees matter for a given interval. In fact, if we look at how we find the subtrees that cover our interval, we'll be using at most two subtrees rooted at each level. The argument for this is not very difficult — if we start with our interval, there may be at most one isolated point on each side of the interval, that if we take it away, the rest of the interval is aligned on even boundaries (the starting point is an even-indexed point). So after we've taken away at most one point from each edge, the rest of the interval can be brought up one more level to the subtrees of two items. And now we can make that same argument recursively at each level — we need to maybe break off one subtree for handling edges, and then we lift our interval to the next layer of the search tree.

So in fact, this tells us we have at most $2 \log n$ subtrees that combine to any query interval.

So we build a search tree on each subtree interval, and we have what we need in order to solve our problem.

Notice that we have to query $O(\log n)$ subtrees, so that means our runtime is now going to increase to $O((\log n)^2)$ — so we pay a little price for this improvement in space.

But now let's talk about the construction time and space. Building a subtree depends on its number of points times $\log n$, so really the question is about the size of all the subtrees we're working with. What is the size of all these subtree data structures that we build? The point is that for all the subtrees at level 1, their total size is n (they cover all the points). And the same is true for the subtrees at level 2, because they're disjoint. At every level the total subtree size is n, so their total is $n \log n$. (A dual way to make this argument is that each point is in one subtree from each level.)

So we have $n \log n$ total space, and this costs $n(\log n)^2$ time to build all of them.

So now we've really nailed the problem. In one dimension it takese you n space to hold the data structure, $n \log n$ time to construct, and $\log n$ to query. Here in dimension 2, everything just increases by a $\log n$ factor.

This immediately generalizes — in dimension d we have $\log^d n$ query time, $n \log^{d-1} n$ space, and $n \log^d n$ construction time. So these are outstanding time bounds, and a natural progression from the 1-dimensional version.

The critical insight was to take the 2D problem and think about how to reduce it to a 1D problem.

> **Remark 27.4.** There is a technique developed called *fractional cascading*, which shaves one log off of all of these time bounds — so in fact, it gives you $\log^{d-1} n$ query, $n \log^{d-2} n$ space, and $n \log^{d-1} n$ construction time. The idea is kind of like persistent data structures — there, as we searched our way down, we identified the right time as we were walking down. Similarly, here as you search down in the first dimension, fractional cascading is able to make some progress to giving your place in the second dimension at the same time.

On Wednesday we'll talk about convex hull and Voronoi diagrams.

§28 November 20, 2024

Last time we started talking about computational geometry. A general approach is to reduce dimension. 1D computational problems are generally things we're used to solving — sorting and lists and things like that. So if we can somehow transform a 2D problem into a 1D problem, we've already got all these techniques for solving it. Last time we saw an example with orthogonal range queries (looking at points in a given query rectangle). We saw a technique for transforming the 2D problem into a pair of 1D problems, where we first filter on x-coordinate and then on y. You have to be clever about this, but we saw how to do that.

§28.1 Dynamic orthogonal range query

Last time we considered the static query structure problem — you build a serach tree in one dimension, and then build sub-search-trees in the other. Once you've done all that, you can answer queries quickly. Assuming a static set of points, you don't actually even need search trees; you can just use intervals in an array. But you do need BSTs for a dynamic version of the problem, where points can be inserted and deleted over time and you want to answer questions about the current arrangement.

§28.1.1 A first attempt

On the surface, solving the dynamic problem seems fine — we know how to do BSTs dynamically. But what issue do we run into if we try to do the obvious thing — when we have a new point that we want to add to our data structure, we simply put it into the relevant binary search trees in both dimensions? Does that work?

At the beginning this seems fine — we've got a search tree on x, so we can certainly add a new point to that. But we used the internal nodes of the x-search tree to find where we build y-search trees. So if we change the x-search tree by inserting or deleting points, we change the structure of that x-search tree, and then we need a different collection of y-search trees.

So the problem is that the insertion changes the x-subtree, which means that we have new internal nodes; and that means we need new y-search trees. Of course, we don't need new y-search trees *everywhere*. If we've built our x-search tree, and we insert a new point somewhere (or delete a point), that will cause changes on the node on the path from the root to this point. And it's these changed locations where we need to compute new y-subtrees. (If we use a rotation-based data structure, then when we insert we do some rotations, and now these points have different children.)

How much this costs us will depend on the sizes of the nodes that we're modifying or rotating as a result of the insertion into the x-subtree. So the cost depends on the sizes of the changed (e.g., rotated) nodes of the x-tree. To put this another way, rotation is now much more expensive than it is when we usually think about BSTs. In a regular BST context, when you perform a rotation, it's just constant-time to update a couple of pointers. But now, every time you perform a rotation, you're going to have to rebuild a substructure in the rotated node, and the cost of that is equal to its number of children (which is basically the size of its subtree). So now a rotation cost is equal to the size of the subtree that is rotated.

So we've finally found something that splay trees are not good for. Splay trees rotate like crazy. And once rotations become expensive, they become a terrible idea.

So instead, we should look for a different data structure. You might want to think about red-black trees. A nice feature is that they only perform a single rotation after each insertion. Unfortunately, that rotation could happen anywhere, and if it happens up high it's very expensive (you could spend linear time for a single insertion or deletion). (This is still better than splay trees, which could be quadratic.)

§28.1.2 Treaps

This is a new data structure problem — how can you build a splay tree when the rotation cost is not constant? Seidel developed *treaps*, which are a very strange combination of BSTs and heaps, and are also randomized. The cool thing about them is that the expected size of the rotated trees is $O(1)$. So treaps are really cool — they perform a small number of rotations and they're almost always near the leaves — so they provide a nice solution to the dynamic tree problem that you need to solve.

There are many similar applications in computational geometry.

But treaps are randomized, so we're not talking about them in this class; you'll have to come to a different class.

§28.2 Sweep algorithms

Continuing the theme of dimension algorithm, the notion of a sweep algorithm is something you apply when you want to construct an artifact — if you take an input and want to produce an output (rather than for a data structure). The idea is you takey our two spatial dimensions, and you turn it into a 1D problem varying over time.

We already played with this with persistent trees, where we had this idea that you look at a single vertical line, and sweep that line across x and watch, as various crossings happen, and update a data structure as you go. When you finish, you'll have constructed this artifact.

So that's the general idea, which will become clear with examples.

§28.3 Convex hull

To make this concrete, we'll talk about one canonical problem, the convex hull problem.

Problem 28.1

Given a set of points, output the smallest containing polytope for these points.

So we have some points, and we need to draw a rubber band around all of those points and have it snap as tight as possible. This is a problem that can be defined in any dimension; we'll concentrate on the 2D version. It's called the convex hull because this hull will be convex (if you have two points inside the convex hull, the line between them also stays inside it). You can also prove it's the intersection of all the half-spaces containing all the points, which also proves that it's convex (the intersection of convex spaces is convex).

This is to computational geometry as sorting is to algorithms. There are 17 different algorithms, each illuminating some idea in algorithm design. We'll see only one of the many.

It's equivalent to sorting in another way as well — if you have a convex hull algorithm (one that enumerates points on the convex hull in order), then you can solve sorting using it. This means you can't beat $n \log n$ time for n points. So the goal is to simply achieve a $n \log n$ time bound. There are many algorithms that do so, using different techniques.

§28.3.1 Sweep line algorithm

We are going to run a sweep line algorithm. To keep it simple, we're only going to output the *upper* hull. If you think about the convex hull, there's a leftmost point and rightmost point; between them there's an upper hull you see from above and a lower hull you see from below. We'll construct the upper hull; you can run the same algorithm upside-down to construct the lower hull as well.

We're going to start by sorting the points by their x-coordinate. We're doing this because we want to sweep a line along the x-axis, visiting the points in order.

So now that we've got the points sorted, we'll think of this in an online perspective. We sweep this line across x, and we discover a point. This point is the leftmost point of the polytope, which means it's on the convex hull — it's an extreme point in linear programming parlance (it optimizes the objective of pointing left), so it's certainly on the hull.

Now we continue to sweep our line across, until we find another point. At the moment, it appears that this point is also on the convex hull. So we keep going, and find another point, and it looks like this is on the convex hull as well; and then another point. Now suppose the next point we find is way up. What have we learned? This tells us that the previous two points aren't on the convex hull —- we can draw a line from

this new point back to the second, and this covers the third and fourth points, so those two points aren't in the convex hull because they're covered by this higher line.

So then we back up our current convex hull until we fix the problem. Now, how do we detect mathematically that this situation has occurred? What's different about this last step that shows us we don' thave the convex hull? The slope of the line is no longer decreasing — in plain English, we turned left instead of turning right as we moved from one point to the next. So if there's a left turn (which we can calculate as a comparison of slopes), that means the point at the front of the hull is not actually on the hull. So we should delete it and back up. Note that when we delete it, we have a proof it's not on the hull. Then we back up to the next point and ask, is this also a left turn? It is, so we delete it and back up again. Now we have a right turn, so we at least tentatively keep it, since we're back to having a convex hull (for now).

So we advance along the points in order. While we see decreasing slopes (i.e., right turns), we continue adding points to the hull. When we see an *increasing* slope (i.e., a left turn), we should delete points from the candidate hull until the problem is fixed.

We've already implicitly proven correctness — we only add points while things look convex, so if we get all the way to the last point and have something convex, then it's the convex hull — because we only delete points when we know they're not on the convex hull (because we saw a left turn proving they're not).

§28.3.2 Runtime

It takes $O(n \log n)$ time to sort the algorithm. Then for the sweep line, we claim that this only costs $O(n)$. Why? We've got all these different stages where we may be deleting some unknown number of points, so why is it just n cost overall? The point is that we spend $O(n)$ add-work and $O(n)$ delete-work — we don't know exactly when during the process we're going to delete a particular point (if ever), but certainly we only delete it once.

Something we weren't explicit about last time but always assume is that our basic geometric calculations are $O(1)$ time — deciding if a point is below or above a line, or whether one slope is bigger or smaller than another, and so on. When you really do computational geometry you have to worry about rounding error and precision and such things, but from a theoretical perspective we assume basic constant-time operations.

Remark 28.2. We should say it's not always a valid assumption that you can do the obvious things in constant time. It's still open how quickly you can determine of two sums of square roots which one is bigger — so if you're comparing the total lengths of some things and other things, that might not be so easy. But all the operations we'll be performing can be done efficiently.

Remark 28.3. There are many other algorithms; you can't do better than $n \log n$, but this is achieved in a variety of ways.

But there is one place where you can give a bound that's better in many cases — an *output-sensitive algorithm*. In sorting you have to output all the points, but in convex hull you only have to output the points on the convex hull; if that's a triangle, then the output is very small. Timothy Chan 1996 gave an algorithm that runs in time $O(n \log k)$, where k isthe number of points on the hull.

This is based on a sampling procedure — you sample a subset of the points, find their convex hull, and throw out the stuff inside. That lets you filter out a lot of points quickly, so you don't actually have to look at all of them $\log n$ times, the way that other algorithms do.

§28.4 Halfspace intersections

There's a related problem of halfspace intersections. In convex hull you're given points and want the line that outlines them. In halfspace intersections, you're given a set of n halfspaces, and you want to find the

intersection. This is basically linear programming in 2D — each halfspace is a constraint, and you apply all the constraints simultaneously, so the intersection is just the feasible region.

If you can actually construct this feasible region, then optimization is very straightforward — the feasible region will be a polytope, and it'll have n sides (each side corresponds to one of your half-spaces), so it also has n vertices. So you optimize by just visiting all the vertices and seeing which is optimal.

So LP in 2D is just a matter of computing the intersection of half-spaces.

How do you do this? There's a very nice duality between points and lines (once you've fixed the origin) — you can construct a map taking the point (a, b) to the line

$$\ell_{a,b} = \{(x, y) \mid ax + by + 1 = 0\}.$$

Essentially, you've got your origin, and if you have a point, you kind of project through the origin; if this distance is d, you end up with a distance of $1/d$; and you make a perpendicular line to that point. This associates every point with a line.

And what's interesting is that if you have some point $(a, b) \in \ell_{q,r}$, that means that $qa + rb + 1 = 0$, but that also means $(q, r) \in \ell_{a,b}$. So this is a duality — if you go from a point to a line, you can also go from the point with the same parameters of the line back to the opposite line. And that means

$$(q, r) = \bigcap_{(a,b)\in\ell_{q,r}} \ell_{a,b}$$

is the intersection, over all points on the dual line, to the line that's dual to this point. Pictorially, if we look at all the points on our dual line, each of them is dual to a line, and all these lines go through this point! So taking a union of points on a line, and then taking the dual, gives you an intersection of lines that leads to the point. So this is a very beautiful duality.

What that means (we're not going to prove it, but you can check) is that the intersection of the half-spaces is the dual of the convex hull of the half-space dual points. So our convex hull algorithm actually solves halfspace intersection as a byproduct, and therefore gives us linear programming in 2 dimensions.

> **Remark 28.4.** Before we abandon convex hull, convex hull in higher dimensions is a very interesting problem. It turns out you can solve it in $O(n^{\lceil d/2 \rceil})$ time for $d \geq 3$. (It's n^2 for 3 and 4 dimensions; the log is gone because the sorting is no longer the dominant term.)

§28.5 Segment intersections

> **Problem 28.5**
>
> You have n segments in the plane, and you want to enumerate all their intersections.

So we want to spit out all the points where two segments intersect. This one we'll barely spend any time on, because we already solved it — we can use the same trick as for our point-location data structure. We sweep line as in our point location with persistent trees. So we have this vertical line that represents our sweep. Notionally, we sweep this line across and notice certain events that happen. You might encounter the start of a line segment, and then we add that to our data structure. Another thing is you might discover two segments cross; that's an intersection, and you output it. The last that might happen is you reach the end of a segment, and remove it.

So exactly as in point location, we have a BST of segments on the sweep line. We have *add* events, *swap* events, and *delete* events. The *add*s happen when we encounter a new segment; the *swap*s happen at intersections; and the *delete*s happen when we pass a segment.

We've already gone through this once with persistent data structures, so we won't say much more.

We want to encounter the segment starts and ends in order, so what we do is we put the endpoints in a heap by x-coordinate; now we're sort of running a time-based simulation, where we pull an event out of the heap, and either it's an addition of a segment to the sweep line or a removal. Also, as segments become neighbors on the sweep line (via insertions or deletions), we add their predicted crossing time to our heap. So we're sweeping our line across; we see our first line and it goes into the data structure, and so does the second. Now we have two lines; we use math to figure out where they're going to intersect. So we know at this time these two segments cross each other. Now as we run our simulation pulling things out of our heap, we may encounter this crossing event, which tells us to output the intersection point and swap those two segments on the sweep line. That may produce new neighbors, which inserts new things into the heap.

(If the intersection isn't on one of the segments — we can also see that by math — then we don't add it to the heap.)

The key fact is that each event uses $O(\log n)$ time for all the necessary updates — to run an event we need to do delete-min from a heap, and then a constantn number of updates of various sorts in the search tree and heap.

And how many events are there? There are n add and delete events, plus k cross events (where k is the number of intersections). This means we end up with an overall runtime of $(n+k)\log n$.

§28.6 Voronoi diagrams

(There is no class on Thanksgiving. After, if there's enough interest there will be some optional lectures. Today will also be the last problem set, which will be due next Wednesday. Because it's the last problem set, David might trickle on some stuff covered on Monday.)

§28.7 Post office problem

Voronoi diagrams are really cool — and algorithms for constructing them are even cooler. The way they're generally introduced is via the post office problem. David used to talk about this in terms of Athena clusters, but no one cares about them anymore.

You've got a collection of points, and you get a query; and the goal is to return the closest point in your set to the query. The points are called *sites* (because we want to distinguish them). And then we get a query point, and we want to find the closest site (under the Euclidean metric, although you could talk about Voronoi diagrams in any metric space).

Of course, if you're just given everything at once, it would take you n time. What's interesting is if you're given the sites in advance, and you get many queries.

Question 28.6. Can we construct a data structure to answer a query quickly?

§28.8 Structure of Voronoi cells

Voronoi diagrams are the geometry of such a data structure. We'll also have to construct them, which is a different algorithmic problem.

Definition 28.7. If we have a site s, then we define

$$\mathcal{V}(s) = \{x \mid s \text{ is the closest site to } x\}.$$

(This is called the Voronoi cell for s.)

Let's start thinking about Voronoi diagrams. First, what's the Voronoi cell of a single point? Of course, that's the entire plane. But what's interesting is if I have two points. Which part of the plane belongs to which point? You can split the plane by their perpendicular bisector — what's critical is the perpendicualr bisector between these two points. This perpendicular bisector defines two Voronoi cells.

The divider is a line, so the Voronoi cell is a half-space; the part of the plane that one point owns is the half-space not containing the other point. What does that tell us about the Voronoi cell for any number of points? If we focus on one site and want to figure out what its Voroni cell is, that's the points closer to it than any othe rsite. So we can consider each site one at a time and draw the perpendicular between them; and the stuff on the far side doesn't belong to me. So every other site excludes a half-space from the cell of this site. Put another way, the set of points that belong to this site are an intersection of the half-spaces that are left alone by all the other sites. So that tells us that the Voronoi cell is going to be a polygon — it's going to be some convex polygon around the site.

Fact 28.8 — The Voronoi cell for each site is a convex polygon.

But let's think about how it's constructed. Let's suppose I take a third point, and look at the Voronoi diagram for these three points. To understand this diagram, we can draw some more perpendicular bisectors. There's one perpendicular bisector, and here's another. Interestingly, those three perpendicular bisectors intersect! Is that just a mistake of the drawing? No — you can think about the three points as forming a triangle, and the three perpendicular bisectors of the triangle sides intersect. Why? What's special about the intersection is that it's equidistant from the vertices of the triangle. The perpendicular bisector is the line equidistant from two points. So if we take a line equidistant from x and y and intersect it with the one for x and z, then that's also equidistant from y and z (by transitivity).

So in fact, this intersection point is the intersection of a *circle* that has these three points on its boundary — this is the center of that circle.

So with three points, we have three perpendicular bisectors that intersect at the center of the circle which is inscribed through those three points.

Now, these three perpendicular bisectors divide space into 6 regions. This seems a bit strange — there should only be 3 regions, since there's only 3 sites. How do we resolve this — where's the Voronoi diagram among these 6 different regions? Each bisector gives you an ordering of distance between two of the points, so for any point, there's an ordering of the three points by distance, and there are $3! = 6$ different orderings. But we don't care about the ordering of the second and third, only the closest point.

So this bisector is saying these two points are equidistant. But once you pass the circumcenter, both are farther away than the third point. So we can just erase this half of the perpendicular bisector. And actually, only half of each perpendicular bisector actually matters — the half where the closest point is on one side of that perpendicular bisector. So we have these three half-lines — these three rays — that together define the closest site for each point.

Now, let's do one more point, because things get interesting. If we have a point all the way outside on the bottom right, you'll notice that the perpendicular bisector for this point — when we had three points, the three Voronoi cells were open. But as we start adding more points, we may actually end up with *closed* regions of the Voronoi diagram. What's a little funny is that when two lines merge, now the perpendicular bisecto ris going to be betwen the two long-distance points.

So you start getting this very rich cell structure dividing the plane. Each cell is a convex polygon, and there's one cell per site, so of course there are n cells. And that defines the shape of the Voronoi diagram.

Ultimately, we're going to want to construct this Voronoi diagram — to figure out what all these line segments are.

Definition 28.9. A place where three perpendicular bisectors meet is called a Voronoi point.

So we have three Voronoi points in our diagram with four sites.

§28.9 Answering nearest neighbor queries

We're going to figure out how to draw this diagram, but first suppose that we *had* it. If we now want to answer the question of what is the closest site, what do we need to decide? We need to decide which Voronoi cell it lies in (which Voronoi cell contains the query point).

So to answer nearest neighbor, we need to find the Voronoi cell containing the query point. So we've turned a problem involving distances into just a problem about which cell of this geometry contains the query point.

And how might we answer this question? We've got an arrangement of segments breaking the plane into polygons, and we've got a query about which polygon contains a given point. That's just the point location problem! So we just need a point location data structure.

And what we remember about the point location data structure is that it had $O(N \log N)$ construction time, took up $O(N)$ space (or maybe $O(N \log N)$), and it takes $O(\log N)$ time to query — we can look back in our notes to make sure David isn't lying.

So we're done, right? All we have to do is construct the Voronoi diagram?

But not quite. Notice that we wrote N in capital letters; why? That's because N is the number of Voronoi points *and* line segments in the diagram. And this is a bit worrisome — the boundary of a particular site could involve all the bisectors with other sites. And if the boundary of every site involves n segments, then the Voronoi segment has n^2 size, so it would actually take us n^2 time to make the point-location data structure and n^2 space to store it.

So that's worrisome. But fortunately that's not a problem, as we'll see next — we're going to bound N.

§28.9.1 Space complexity

The first thing we'll notice is that the Voronoi diagram is a *planar graph* — the sites don't matter anymore, it's all about the Voronoi points and the line segments that connect them. If we think about these Voronoi points as vertices of a graph, and line segments connecting them as edges, then we have a planar graph — none of the edges cross each other. (The edges are the bisectors, and any time the bisectors cross, we've made a Voronoi point.)

We may be worried about the lines going to ∞, so what we'll do is add a vertex at ∞ for all the unbounded lines. If you're worried about the fact that they're going in different directions, imagine we do this on a giant globe, so all the lines that stretch out to ∞ are wrapping around to the far side of the globe, and we can have them meet up there. Then every line segment is between two actual vertices; we don't have any dangling vertices.

Now that we have such a nice planar graph, we can apply Euler's formula:

Fact 28.10 — If you have a connected planar graph G, then

$$n_v - n_e + n_f = 2$$

(where n_v, n_e, and n_f are the number of vertices, edges, and faces.

Faces are the region enclosed by a closed loop of line segments, which in our graph are just the Voronoi cells.

We already know that $n_f = n$ is equal to the number of sites. But we're worried about the size of the graph, which involves vertices and edges; so we'll have to bound that.

Proof of Euler's formula. We'll do this by induction. Note that 2 is an invariant — we're trying to show $n_v - n_e + n_f$ is a constant. So we'll show a way to simplify the planar graph in a way that we don't change the sum, and eventually we'll have simplified it far enough that the value of the sum is obvious.

If we have 0 edges, then this is obvious. Why? Then we'll only have one vertex (because we assumed the graph is connected), and there will be 1 face; so that gives a sum of 2.

We can also observe that if we have an edge with both ends at the same vertex (i.e., a self-loop), then if we delete this edge, we're getting rid of one edge, but we're also getting rid of the face inside the edge. So this also doesn't change our sum (we get -1 face and -1 edge).

Another possibility is if I have an edge with two different endpoints. In that case, I *contract* that edge. This decreases the number of edges by 1. But it also decreases the number of *vertices* by 1. So it again leaves the invariant unchanged.

So by this sequence of transformations (contracting edges and removing loops), I simplify and simplify until I get down to the most boring possible case, where the invariant is obvious (none of my transformations change the number). □

As stated, we have $n_f = n$.

> **Claim 28.11 —** Every Voronoi point has degree at least 3.

We can see this in the picture, so it's obviously true. Why did we say at least 3 — how could we end up with a Voronoi point higher than 3? It's possible to have four points that are all on the same circle. This is a degenerate case — you'll have to have your numbers exactly right (any small perturbation in the position of the points will change this). So it's often convenient to assume for simplicity that there are no four equidistant points (you can achieve this by tiny random perturbations, or lexicographic comparisons).

But why can't it be 2? What produced the Voronoi point is that it came from the intersection of three perpendicular bisectors; they're all there, so they all have to contribute to the degree.

Since every Voronoi point has degree at least 3, now we can do the trick where we count things in two ways. We can sum up the endpoints of all edges — this is $2n_e$ (we're counting edge-endpoint pairs, by summing over all edges). Another way to do that counting is by summing over all vertices — we can count, for each vertex, the degree of that vertex. So we have

$$2n_e = \sum \deg(v) \geq 3n_v$$

(note that we're counting the point at ∞ as well in n_v). If we rewrite this, we have n_e here, and we cna plug in Euler's formula, which says $n_e = n_v + n_f - 2$; then we get

$$n_v + n_f - 2 \geq \frac{3n_v}{2}.$$

But remember $n_f = n$. Then we can move things around to get

$$2(n_v + n - 2) \geq 3n_v,$$

and subtracting off $2n_v$ gives

$$2n - 4 \geq n_v.$$

So we've shown the number of Voronoi points is at most twice the number of sites.

This also means that the number of *edges* is also $n_e = O(n)$. And now everything is wonderful. The Voronoi diagram reduces near neighbor to the problem of point location in the Voronoi diagram, and we've shown the size of the Voronoi diagram is linear in the number of sites. So the N from before is actually n; and

we're able to solve this near neighbor problem basically optimally (with $n \log n$ time to construct, and $\log n$ to answer queries).

All that's left is actually constructing the Voronoi diagram. Our input is just the sites; we need to build this whole structure so that we can give it as input to the point-location data structure. That's what we'll do on Friday.

§29 November 22, 2024

§29.1 Intro – constructing the Voronoi diagram

Today we'll cover Voronoi diagram construction. Last time, we introduced Voronoi diagrams — you start with n sites, and the Voronoi diagram constructs around them n convex polygonal *cells*, each owned by a site, consisting of all the points which are closest to that site. The boundaries between cells (which we call edges) are parts of perpendicular bisectors of sites. And the vertices are intersections of 3 (or more, in the case of degeneracy) edges. This means these vertices are the center of a circle inscribed through three (or more) sites.

Last time, we showed there are $O(n)$ edges and vertices, and exactly n faces (those are just the regions of the Voronoi diagram). So we were able to do 'nearest neighbor' via point location, which takes $O(n \log n)$ time to construct given the Voronoi diagram, and supports $O(\log n)$ query time. That's where we left off last time — the Voronoi diagram yields a great approach to answering nearest neighbor queries.

But to do that, you have to start with the Voronoi diagram; we're only starting with the sites. So today we'll talk about how to construct the Voronoi diagram from the sites. There are many different algorithms for this, using different concepts. For example, there's a randomized incremental construction where you add one site at a time and update as you go.

§29.2 Sweep line approach

We are going to use a sweep line approach — taking a 2D problem and reducing it to a kind of 1D problem, and thinking of the second dimension as time — thinking how the Voronoi diagram grows over time as we sweep a line across the plane. To be consistent with the literature (and make some pictures easier to draw), we'll have a sweep line that drops vertically from $y = +\infty$ down to $y = -\infty$. And we're going to build the Voronoi diagram behind (i.e., above) the sweep line.

So essentially, we'll have the sweep line, which is descending. Below the sweep line there are some sites. Above the sweep line there are also some sites. And we want to have the Voronoi diagram of the area above the sweep line already built. So up here we have our Voronoi diagram constructed; we want to run the sweep line down and grow the Voronoi diagram as the sweep line falls.

§29.3 The beach line

There's one immediate problem with this approach — the Voronoi diagram above the sweep line is not fully determined by the area above the sweep line. How can something below the sweep line change the Voronoi diagram? Imaigne we've got this whole Voronoi diagram, and there's a point only slightly below the sweep line that owns a bunch of the region above it. So we can't say that part of the diagram is determined.

So then, given that this is the position of the sweep line, what area above it can we be confident has been fully worked out, just based on the sites above the sweep line? Intuitively, we've got some site up here, and it clearly owns everything in a small circle around it — nothing that happens below the sweep line could

possibly change that. But where could things be changed? What's the boundary between what is known and what is unknown?

If you think about it, if we're at this point and we've got this line, let's assume for now this is the only point. What part of the space above the line am I sure belongs to this point? The boundary is the set of points that are equidistant from the site and the sweep line. And the set of points equidistant from a point and line is a *parabola*.

So for a given site, the set of points equidistant from the site and the sweep line form a parabola. And we can conclude that nothing above the parabola is changed by anything yet below the sweep line — anything above the parabola is closer to this site than to the sweep line .It might even be closer to some other site, but it certainly won't be owned by anything below the sweep line, because this site is better than anything below the sweep line.

So we can actually go and draw a parabola associated with each site. So there's this collection of parabolas defined by the sites and the sweep line. And nothing above one parabola is going to change, so nothing in the union of the stuff above the parabolas is going to change.

So this union of parabolas is what we'll call the *beachline* (the reason is as the sweep line goes down, all these parabolas also come down, and they look a bit like ripples where on the beach waves come in and bulge out in certain spots). So this beachline is going to be descending as the sweepline does. And it's the region above the beachline that has been determined already (i.e., determined by the sites above the sweep line).

So if we run the sweep line algorithm, we'll need to evolve or grow our Voronoi diagram as the beach line descends.

Let's think about some other features of the beach point. Where do two parabolas intersect? A pair of parabolas, if they're intersecting, that means you're equidistant from the two foci of those parabolas, because each is at the same distance as the sweep line. So the intersection of the two parabolas is on one of the edges of the Voronoi diagram. This isn't exactly true — not all intersections are edges, because not all perpendicular bisectors are in the Voronoi diagram (and similarly certain intersections of parabolas are not relevant). But at least in our picture, we have a couple of sites, and as the sweep line descends, their intersection of parabolas will also descend and will actually trace out this edge of the Voronoi diagram.

§29.4 Descent of parabolas

So now let's fill in some numbers. Let's look at the descent of one parabola. What does this mean? We have a sweep line at $y = t$, and t is slowly decreasing. We have a parabola defined by the equidistance formula

$$(x - x_f)^2 + (y - y_f)^2 = (y - t)^2$$

(where (x_f, y_f) is the focus — the left-hand side is the (square of the) distance to the foci, and the right-hand side is the vertical distance to the sweep line). If we fix x and take the derivative with respect to t to understand how the parabola descends as we move the sweep line, x is fixed, so we get

$$2(y - y_f)\frac{dy}{dt} = 2(y - t)\left(\frac{dy}{dt} - 1\right).$$

If we move things around, this lets us conclude that

$$\frac{dy}{dt} = \frac{y - t}{y_f - t}$$

(this is the rate at which a point on the parabola falls as the sweep line falls).

What does this mean? Now that we have this rate of descent for one parabola, we can talk about multiple parabolas and compare their descents. You'll notice that the higher y_f is, that makes the parabola descend more slowly (larger denominator means the rate of descent is less). This is going to help us understand when parabolas might pass each other as the sweep line moves down.

§29.5 Site events

Let's think about how we end up with two parabolas. We start off with one site, and we've got our sweep line and this parabola being traced out. And we suddenly encounter another site on the sweep line. How do we have to change the beach line as a result of encountering this other site? Well, I have to make a parabola for this new site. At the moment of contact, the parabola is degenerate — it's just a line straight up. But soon after I pass the site, I'm going to end up with a very skinny, narrow parabola. You'll notice this parabola actually intersects the other parabola twice — it creates three separate sections of the beach line (two involve the same parabola, but they're still topologically distinct).

This will be called a *site event* — when the sweep line hits a site. This is going to create a new parabola, which may split other parabolas.

Here's an interesting thing — we said the intersection of these two parabolas is on the perpendicular bisector between the two points. So if we trace out that perpendicular bisector, the two intersections of the parabolas are going to trace out the perpendicular bisector in opposite directions. That means one is tracing an intersection downwards, but the other is actually tracing an intersection upwards (unless the line is perfectly horizontal, which is a degenerate case we won't worry about).

So we get this site event, and that introduces a line into the Voronoi diagram that we didn't know about before.

§29.6 Circle events

We've just seen a situation in which a Voronoi edge gets created or appears at this intersection point. What causes an edge to *disappear*? When does an edge stop? This edge is going to grow downwards as the beach line moves downwards; when is this edge going to stop? That happens when it's overtaken by another parabola. What happens at that moment of intersection?

So we've got this edge coming down because it's at the intersection of two parabolas. When it gets overtaken by another parabola, what happens in the Voronoi diagram itself? When an edge stops, it stops at a vertex — a vertex occurs when you have an intersection of three of these Voronoi edges. And how does this intersection of three Voronoi edges happen? This means you have two separate intersections of parabolas; each of them is tracing out a Voronoi edge, and at a certain moment as the sweep line is descending, you end up with all three parabolas intersecting at the same place. We've got to check the geometry, but from the picture you can see that at this moment of intersection, you had three parabolas tracing out two lines; and these lines collide. As you descend further, you only have one line — one of the three parabolas actually fell behind the other two, and is no longer a part of the beach line. We've seen creation of edges, but this *reduction* of edges is a result of a parabola *vanishing* from the beach line.

We're going to call this a *circle* event — when we have three parabolas that intersect at one point. The reason for the name is that if three parabolas intersect at one point, then that point is a Voronoi vertex, which means it's the center of the circle inscribed through the three corresponding sites to those parabolas.

§29.7 Ruling out other events

And it turns out these are the only events that occur — the only things that change the topology of the beach line and the Voronoi segments being traced out. You can have a site event, which creates a parabola whose focus is at the sweep line; that's the only way parabolas get created. And you can have a circle event when one parabola falls behind two others.

Let's sanity-check that. Can it happen that you've got three descending parabolas, andn you sort of initially only saw two of them, and then the third one kind of overtook them and you got an appearance of that third parabola which was previously invisible? The point is that this can't happen because higher parabolas

descend more slowly. In our picture, the middle parabola has a higher focus, so it'll be descending more slowly than the two parabolas from the outside; so it'll be overtaken by the other two, and it'll disappear from view as opposed to appearing in view.

(The parabola that drops out is always the middle one, not necessarily the highest one.)

§29.8 Data structures

That's the geometry; now let's think about data structures. We basically need to run a simulation of this line sweeping down. But we don't want to run in continuous time — we need to use a discrete event simulation. So we want to maintain a priority queue of upcoming events, and we'll want to take the soonest event off the priority queue; and we want to use this to predict future events. We did this in the point-location data structure where we detected a change in the ordering as the result of some event, and then we had two new lines that were neighbors, and we predicted where they were going to intersect and put an event corresponding to that intersection. We'll do the same with the Voronoi diagram — we'll try to track the site events and circle events. We just need to be sure they're all put into the priority queue early enough that we'll be able to pull it out of the queue at the time I'm supposed to.

So I need to be able to predict those upcoming events. In terms of data structures, what we're going to use is a *tree* on these parabola segments of the beach line. In our picture we have three parabola segments, so those are going to be 3 leaves in our tree, in the order in which they appear on the beach line.

And this also gives us the neighboring segment intersections, which are the Voronoi edges.

Now as the sweep line descends, the beach line descends, but the order of these intersection points doesn't change — the order of these parabola segments does not change.

We need to deal with site events — how do we update our collection of parabolic segments when a site event happens? We've pulled that site event out of the priority queue (which is ordered by y-coordinates of events). So at a certain y-coordinate, the sweep line hits a site. What do we have to do in order to update our beach line in this data structure?

A site event means there's a new parabola, which is initially a line but widens out; and it's going to cut some parabola that's currently on the beach line. So we want to replace that one parabolic segment on the beach line with three — the outer two on the original parabola, and the inner one from this new parabola. So we need to cut a parabola into two segments, and insert a new parabola in between.

How do I find the parabola that I need to cut into two segments? This is why we're using a search tree, but what do I search on? I have an x and y coordinate for the new site, but as the sweep line moves down, the x and y coordinates of these parabolas are changing, so how do I figure out which parabola is at this moment at the x-coordinate of my new site?

Each of our Voronoi edges is a line, so it has a linear equation, and these are being traced out. So we have all these linear equations, and we can ask, for my new site, where is it relative to these lines? This is the same sort of thing we did with predicting line crosses in point-location, but more complex.

So we've got this tree on parabola segments; and the question is, as we descend the tree, do we go to the left or right? (You can think about this as binary searching on the parabolic segments.)

All the parabolas are somewhere over the new site; we want the one that's lowest.

We've got these lines being traced out by the intersections of the parabolas; we can write down their equations and such. So given where the sweep line is, we can identify the x-coordinates of these intersections, and we can binary search on these x-coordinates to figure out which pair of intersections we're between, which will identify the parabolic segment.

So we do a binary search (tree) through these intersection points of neighboring parabolas to find the parabola segment that needs to be cut.

The nice thing about site events is we know at the start when they'll happen — at what coordinate the sweep line reaches the site event.

§29.8.1 Circle events

What about circle events? These have to be predicted — a circle event occurs when two Voronoi edges collide to turn into a single Voronoi edge. And just as we did with the point location data structure, in order for two edges to collide in this way, sometime before that they have to be neighbors in the ordering.

So in order to prepare for these upcoming circle events, any time two edges become neighbors in the ordering (due to insertions and disappearances of parabolas), we do a computation on those two lines to figure out at what value of the sweep line that intersection will occur. And we put the circle event into the priority queue, associated with that position of the sweep line.

So a circle event must happen between two adjacent Voronoi edges growing on the beach line. So when two edges become neighbors on the beach line, we compute the moment that they will intersect, and put that event into the priority queue. If we pull that event out of the priority queue, then we can execute what hte circle event tells us to do — to take the middle parabola out of the beach line (it falls behind the other two), which makes the other two parabolas into neighbors. This may create a new neighbor relation between intersection points. So when we have a circle event, it may cause us to insert some new circle events into the priority queue. But these are circle events that happen later than the one happening right now. So we can maintain the invariant that we'll always have events in the priority queue in time to pull them out, so we can execute events in the right order.

§29.9 Time analysis

To summarize, we maintain a BST to let us hone in on the right parabolic segment. We'll sometimes need to insert a new one or delete a vanishing one; each of these operations takes $O(\log n)$ time.

So we have this priority queue, and we have n site events inserted at the start of the process. Then we run time forward. On a site event, we insert a parabola, and inserting the parabola takes $O(\log n)$ time.

(Running time forwards takes $O(\log n)$ time per event, and inserting the n events at the beginning takes $O(n \log n)$ time.)

On a site event, we insert a parabola; this can create $O(1)$ new neighboring Voronoi edges on the beach line, so as a result, we might insert $O(1)$ new circle events into our queue. But inserting each of those circle events takes $O(\log n)$ time, so this overall takes $O(\log n)$ time to update the priority queue with the new events.

A circle event is the same — we need to update our parabola tree, which takes us $O(\log n)$ time. And we need to insert $O(1)$ new circle events, which also takes us $O(\log n)$ time. So whatever event we process out of the priority queue, it costs us $\log n$ time to process that event, remove it from the priority queue, update the search tree, and insert any new events.

So the only question remaining is, how many events do we process? And we process exactly n site events (we hit every site exactly once). What about circle events — how many circle events are there? Circle events correspond to vertices of the Voronoi diagram (i.e., Voronoi points). And there are $O(n)$ vertices, so there's $O(n)$ circle events. That means the whole construction of the Voronoi diagram takes $O(\log n)$ time.

§29.10 Higher dimension

Computational geometry is generally focused on low dimension, but not just dimension 2. So you have higher dimensional Voronoi diagrams too. In 3D, some of what we said is still true — every site will own a

particular Voronoi cell, which will still be convex. And so it's a polytope in higher dimensions. And there are very interesting questions about how to construct these higher-dimensional Voronoi diagrams.

One interesting place you see Voronoi diagrams is in soap bubbles. If you do experiments that create lots of soap bubbles pressed up against each other, they basically map out a Voronoi diagram.

§29.11 Delaunay triangulations

We should also mention that the Voronoi diagram has an interesting geometric dual (or graph dual). Last time we talked about duality with respect to convex hull vs. halfspace intersections. If we draw a Voronoi diagram, you can create a dual structure. What you do is, you take every pair of sites that are adjacent (their cells share an edge), and you draw an edge betwen those two sites. Why is this a dual? Each face (which corresponds to a site) becomes a vertex of a new graph. And in a dual sense, it creates faces that correspond to the vertices of the old graph. So faces and vertices swap roles. Meanwhile, each edge turns into another edge (it sort of turns sideways from itself).

If you look at this dual graph, all the faces of this dual graph are triangles. Why? If you're not degenerate, then every Voronoi point has degree 3. So it has three incident edges, which means when you take the dual, it'll have three edges around its face.

So this dual graph of the Voronoi diagram is a *triangulation* — a division of the entire space into triangles. And it's a very special triangulation known as the *Delaunay triangulation*. It's in a formal sense the very best possible triangulation — you start with the sites, and I can say, I want you to use them to create a triangulation of the plane where the vertices of the triangulation are these sites. You might need to do this if you're doing a fluid dynamic simulation, and you've got a bunch of points whre you need to measure velocity of the fluid. So you need to create this triangulation to make the continuous plane discrete for your simulation. What's nice about the Delannay triangulation is that it has the 'un-skinniest' triangles. What's bad in a triangulation is if you have a very long, skinny triangle — because in reality things could be very different at the two ends. So you want triangles with large angles to support good simulations. This is also used in computer graphics when you want to render things — you create a triangulation of your shape.

The Delaunay triangulation maximizes the minimum angle of any triangle in the triangulation — so it has the best 'aspect ratio.'

Our Voronoi diagram algorithm is also a Delaunay triangulation algorithm — all you need to do is build it and take the dual. So that's another wonderful outcome.

> **Student Question.** *How do you prove this?*
>
> **Answer.** That's a lot more work. But it comes out of the fact that these cells sort of represent what's closest to every site — if you used a different triangulation, you'd get a point associated with a site that's closer to another site, and from that you can infer you've got a funny-shaped triangle.
>
> There's a condition that has to do with the way these different inscribed circles overlap each other.

§29.12 External memory algorithms — the model

Now we'll talk about external memory algorithms. Today we'll give an introduction, and we'll do a bunch of it on Monday. There's one problem about this on the problem set so that we have a little bit of experience; that will make sense on Monday. We won't teach on Wednesday because of Thanksgiving.

The narrative here is somewhat similar to online algorithms and paging. In modern computer systems you have small fast memories and large slow memories. Because you can't fit your data into the small fast memory, you need to keep it in the large slow memory and access it in small amounts for the purpose of

your computation. Because access is slow, it's a terrible idea to just fetch one item from your slow memory; instead they give you data in *blocks* of many items or words.

So the external memory model assumes that you have a small fast memory for M items, and an infinite slow memory. And you can fetch from the slow memory in contiguous *blocks* of B items (or words, or whatever you want to talk about). We're going to talk about problems of size $N \gg M$.

In our online algorithms narrative we said, well, we're just going to run our favorite algorithm, which does memory accesses, which are references to things in the slow external memory; and we have a paging algorithm that fetches in blocks with the thing we want, and carefully manages what's in the main memory (we used k for paging instead of M) and we could sort of compare the performance of our algorithm to OPT. But this was based on assuming there was a sequence of requests we didn't control — it just came because of how the algorithm ran. If you run an algorithm that's unaware of this external memory behavior, it might do something very silly.

So the idea behind external memory algorithms is to focus on the fact you have slow external memory, and try designing algorithms that minimize your accesses to that slow external memory.

As theoreticians, we want a nice, elegant model. So in order to define parameters or an objective for this goal, we'll say that each external memory access costs 1 (just like in paging). We'll also say that computation in the fast memory is *free*. So we're going to focus entirely on the cost of fetching things, and we'll assume you're free to run an exponential time algorithm instantaneously as long as you're only accessing what's in fast memory. Of course this is a cow-is-a-sphere simplification of what's really going on, but it lets us focus on one interesting question — how to be clever in our use of external memory. Once you answer this you may also try to be clever about how we do the computation in the fast memory, but we want to solve one thing at a time.

§29.13 Basic operations

Let's talk about some basic operations and how much time they take. If I want to scan an array of N items stored in external memory (maybe I'm looking to match a certain value or something like that), how long does that take me? It's N/B — we get B items at a time with each block read, and we need to do N/B of those block-reads.

If we want to add two vectors of length N, it's pretty much the same thing — we just scan the two vectors, take the first block of each vector into memory, add them, write that out to memory; so that's two reads and one write for each output block, which is also $O(N/B)$ reads and writes.

> **Remark 29.1.** Note that we pay for both reads and writes — an 'access' refers to either.

Similarly we can reverse an array — we read a block from each end, swap and reverse them in main memory, and write the swapped blocks in the appropriate places.

> **Student Question.** *Do we assume $M \gg B$?*
>
> **Answer.** Yes — we assume that, and that we have an integer number of blocks. (The algorithms we described so far only require $2B$ space, but usually M is much larger.)

§29.14 Matrices

First, how much time does it take me to add two matrices (we'll say they have dimensions $N \times N$)? This is N^2/B — we can just treat it as two vectors (of length N^2) and add it the same way.

But how long does it take me to *multiply* two matrices?

Now the answer depends a lot on how we're storing the matrices. Traditionally, we might store matrices in row-major order — we've got matrix A, and its first row takes up some number of blocks, its second row takes up some blocks, and so on; and we have another matrix B, which is laid out the same way. How long is this going to take to do a multiplication, if we store things this way?

Well, what happens if we externalize the standard matrix multiplication algorithm? To produce the top-left corner entry, we multiply a row of the first matrix by a column of the second. To read the first column, I need to read one block for each row. And there's N rows. So I actually have to do N reads through the second matrix. It's only N/B through the first, but that's dominated by the N reads for the second matrix.

So it's N reads per output value. And of course there are N^2 output values. So that means I need to do N^3 reads. (We only need to do N^2/B writes, but we have to do a crazy number of reads.)

How might we fix this? An easier thought might be, what killed us is the second matrix is stored in row-major order. If it were in column-major order, what would that do? Then you only have N^3/B, which immediately points out how important it is to think about storage.

But there are two problems with this. One is that this may work great for this calculation, but what if the next time we want this to be the first argument instead of the second? So we want it in column-major order sometimes and row-major other times. This requires us to solve the problem of transposing efficiently.

That's one direction you can go down, but even if you do that, it turns out this is inefficient — even if you have a nice layout for the second matrix, this algorithm ends up reading the same block many times for different computations. The first row needs to be read for each of the entries in the top row that's being output. So if you do that, you're basically making yourself read this top row N times, even though this top row is not very large. So you're wasting a lot of reads.

You can do much better if you think about breaking the matrix into blocks. Instead of running a general row major solution, you can think about submatrices. We'll take each matrix, and think about it as a bunch of smaller (square) matrices. For example, we might have

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} X & Y \\ Z & W \end{bmatrix}$$

And in order to compute the top-left entry, that's $AX + BZ$.

What size should these submatrices be? If we can fit one block of each matrix into our fast memory, then we can do the entire multiplication of those instantly, and just store the result.

So we'll split our matrix into submatrices of size $\sqrt{M} \times \sqrt{M}$. (We'll pay some constants here.) This means I can read a whole block into my main memory from each of the two matrices I'm multiplying together. Even if there are many blocks, I can sort of accumulate the value of the output — I multiply AX and hold that in main memory, then I bring in BZ and add that to what was in memory; and so on. So I only need to keep three blocks in memory at a time in order to compute an entire output block.

That means the main matrix is $(N/\sqrt{M}) \times (N/\sqrt{M})$ of these blocks. And that means if we think about the standard matrix multiplication algorithm, it performs n^3 products. So here we'll be doing $(N/\sqrt{M})^3$ block products to produce the entire output. And each of them requires just M/B reads — we chose the size of a subblock to be M, so reading the whole block in costs us M/B. And that means the overall runtime is

$$\frac{M}{B}\left(\frac{N}{\sqrt{M}}\right)^3 = \frac{N^3}{B\sqrt{M}}.$$

So you can see that by using this clever layout, we're able to outperform even what we'd get if the second matrix were in colum-major order — we're doing a much better job of only reading data we need a few times, instead of reading the same data over and over again. We get a speedup by $\sqrt{M}$. And in modern computers, M could be $64 \cdot 10^9$, so this is a big deal. And all we needed to do was use this clever layout or algorithm for the order in which we compute things.

Note that we can still use a row-major layout for the matrices (in our blocks); but we're not reading an entire row all at once, instead parts of many rows that form a block. This also does away with the need to transpose. Both of these matrices can be in row-major order, because we're just reading blocks rather than entire rows.

§30 November 25, 2024

Today we'll finish our taste of external memory algorithms. We have large slow external memory, and small fast internal memory. We access external memory by reading blocks of size B. We have an internal memory of size M, and a problem size the number of items, generally N. Unlike online algorithms where we're trying to figure out how to serve an arbitrary series of requests, here we're designing algorithms where we decide which specific pages to request; that gives us flexibility to plan a schedule that can be served more efficiently than an arbitrary one.

We looked at some basic problems, scanning and reversing an array, and we talked briefly about matrices. Addition is easy (it just involves scans through the data, which takes $O(N/B)$ time). For multiplication, we saw the natural naive algorithm (where matrices are in row-major order and you use the standard algorithm of multiplying the appropriate row by column to get an output value) takes N reads to get the N values out of the column you want, so that takes N^3 reads of external memory. You can be smarter by transposing the second matrix first so it's in column-major order, and now scanning one row of the first and column of the second costs you N/B, reducing cost to N^3/B. But that's still not taking advantage of your fast memory. If you improve the locality of reference in your matrix multiplication algorithm, you can improve runtime. We divide the matrix into blocks of size $\sqrt{M} \times \sqrt{M}$, and produce one output block by multiplying a row and column of input blocks (you just scan through one row of input blocks in the first matrix and column in the second, and multiply and accumulate into an output block). The advantage is you use one read of a row to produce many output values, rather than rereading the row for every single output value. This improved runtime from $O(N^3/B)$ to $O(N^3/B\sqrt{M})$, demonstrating that you can take advantage of your internal memory to make your algorithm faster and faster.

Today we'll continue on this vein looking at other basic data structures and algorithmic problems.

> **Student Question.** *What are the relative sizes of B and M?*
>
> **Answer.** In general B is reasonably small (e.g., kilobytes or megabytes), and M is quite large (e.g., gigabytes). However, N might be on the order of terabytes, so it won't fit into M, which is why we need the external memory. (But we assume you can fit lots of blocks in memory, so we ignore additive factors. Also, above we're actually using $O(M)$ memory, since we need to keep a few blocks around, but we won't worry about constant factors.)

§30.1 Linked list data structures

There are millions of uses for linked lists, but what if the linked list doesn't fit into main memory and has to be stored in external memory?

We want to support insert, delete, and traversal of the linked list. If we just treated external memory as a great big version of internal memory and maintained a linked list there, then when we allocated a new node we'd find some block with free space, throw it in there, and update some pointers from the previous element to the successor. This involves reading a few different blocks — if you're sitting on the predecessor where you want to insert, then you need to store the new node somewhere, update a pointer from the predecessor, read the predecessor to get the successor, store the successor, and update the predecessor item. So it takes you $O(1)$ work to do an insertion and deletion.

But what about scanning — how many block reads are you going to need to do in order to traverse k items in the linked list, starting in one place and following k pointers? This costs you k for k items. And that's inefficient — you have B items in a single block. It seems like you should be able to achieve something like k/B. But if you're following a pointer to a different block on every step, it costs you k rather than k/B.

When we talked about scanning an array, that really does cost N/B because the array is sorted.

So can we achieve the best of both worlds — have N/B scan time, but be able to insert and delete in arbitrary places in the list?

It turns out that we need something kind of in-between the dense sorted array and the random jumping around of a linked list. What we're going to do is we kind of want to be able to insert items where they belong in the sorted order, so we can traverse in sorted order. But to be able to do that, we need to have space where we want to insert these items.

So we're going to arrange for our blocks to not always be entirely full.

The idea is to keep an invariant that each block is going to have between $B/2$ and B items. And even though we want a linked list, we're going to store it as a sorted array, so we don't have to jump around all over the list to pointers. As long as each block has at least $B/2$ items, we'll only be spending $O(k/B)$ time to scan the items. So that gives us the speed we need.

How do we achieve that? Well, kind of by brute force. On an insert, if there's room, we're done — we can just put the item into a block that isn't full. What should we do if we have a full block and we want to insert an item into it? You can split it into two half-full blocks, and then add the item. The point is that we are going to maintain sorting within blocks, but the blocks don't have to be stored in any particular order on the disk — we have random access into the external memory. So where the next block is in external memory doesn't matter. So if one block is full, we just allocate another block, put the second half of items in there, and point to the next block. So we're keeping a linked list of *blocks*, where each block points to the next block in the sorted list of items; but the individual items themselves are sorted.

And what about delete? If there are more than $B/2$ items, we can just delete the item from the block. But what if we have exactly $B/2$ items? Now removing it is going to violate our invariant. And we care about violating the invariant because if we end up with too few items in a block, then we're going to have to read too many blocks to get through the items.

So how do we preserve the invariant of having each block be half-full, if the current block is exactly half-full? The idea is to balance with the next block.

So we read the next block. There are two interesting cases for what to do with the next block. If it also has exactly $B/2$, then we merge them into one block. If it has more than $B/2$, then we just redistribute the items betwen the next block and current block, to end up with at least $B/2$ per block.

So you can see that each insert and each delete involves touching a constant number of blocks, so it's still constant time, just like the naive approach. But we're able to preserve the invariant of at least $B/2$ items per block, which means we're scanning efficiently at a cost of about $1/B$ per item scanned.

> **Student Question.** *Do we have a pointer to each item in the linked list?*
>
> **Answer.** No — the linked list is bigger than internal memory. But the way linked lists work is generally that you have a finger — you traverse the linked list until you find what you're looking for, and if you want to perform an insertion at the finger, then you can do that in $O(1)$.

§30.2 Search trees

We can implement a binary search tree in external memory in the obvious way — put pointers in the nodes, and follow those pointers in the normal fashion. But this is going to be inefficient. A binary search tree

takes $O(\log N)$ time to do various inserts and deletes and search operations. And you can already see this is not efficient because each step in the binary search tree involves looking at a single comparison item, but when we read a block, we get B items. So it's foolish to only consider one of them.

So what's the enhancement we should use in order to improve our operation time in this external memory model? If a single read gives us a block with B items in it, how can we take advantage of that?

You can increase the degree of the tree — with a degree B tree, you might hope to get a runtime of $\log_B N$ instead of $\log_2 N$.

If you want this sort of runtime, you need the search tree to stay balanced. We know a lot about maintaining balance in binary search trees, but haven't really discussed this in a B-ary search tree. So we'll now describe one particular structure that achieves balance, called a *B-tree*.

The implementation of a B-tree maintains an invariant that all of the leaves are at exactly the same depth, which is $O(\log_B N)$. How does it do that? Well, we use the same idea as we used from linked lists. We're going to ensure that every node, both leaf and internal, has between $B/2$ and B items. This is not actually going to be a degree B tree, but a degree between $B/2$ and B tree. But that means the depth is less than $\log_{B/2} N$, which is $O(\log_B N)$.

So let's talk about the operations of insertion and deletion. It's clear how you search — you just walk down the tree reading blocks, and look at the items in the block to decide what child to read next.

How do we maintain this invariant on insertion? (We'll also keep all items at leaves, and send copies upwards as the splitters to help us decide where to descend in the tree.)

When we insert, we walk down the tree until we find the leaf where the new item belongs. So we find the target block, and if it's not full, we're done — we can just put the item there.

What do we do if the target block *is* full? Well, we can't fit everything in one block, so we need to split it into two blocks, and send a splitter between them up to the parent node. (We had only one block, which was between two items in the parent node; now we're breaking it into two blocks, so we need to add another splitter to the parent node.)

What could go wrong? Maybe the parent node already has B splitters in it. What should we do then? If the parent is full, we should split the parent node and send a splitter between the two to the grandparent. And we just keep cascading these splits as long as we encounter full nodes.

At some point along the way, we'll find a not-full node and put the splitter there and stop; or else this cascade will continue all the way up to the root. And if the root is not full, great. But what if the root is full? If we go all the way to wanting to split the root, well, how can we split the root? That represents the idea that we want to turn the root into two pages, and we need to start by making a decision of which of the two pages to read. So how do we do that? We'll just add another layer to the B-tree.

So this is actually how the B-tree gets taller — we're always keeping the leaves in the same bottom level, and the tree gets fuller and fuller in its internal nodes, until the root gets filled up. And at that point, we add another layer to the B-tree to hold a new, higher root.

But now the root only has 2 children. So we actually require every node *except the root* to have at least $B/2$ children. This is fine, because we only have one root; so this adds just 1 to the cost of the search, which doesn't matter.

So that's insertion. What about deletion? It's kind of the same in reverse. The first step is straightforward — we walk down to a leaf. Now we've found the leaf where we need to remove an item.

The easy case is when it has more than $B/2$ items; then we simply remove this one. However, if the leaf has *exactly* $B/2$ items, then removing one is going to violate our invariant. So what should we do? With linked lists, when we found that a block was half-full, we looked at the successor leaf node. If it has more than $B/2$ items, then we can move some over to this node, so that both of them have $B/2$ items. Note that this means we need to change the splitter between the nodes, so we have to replace the splitter in the parent.

So we check the successor. If there are more than $B/2$ items, then we can redistribute and update the parent splitter. On the other hand, if we have exactly $B/2$ items, then just as in the linked list case, we can merge two leaves into one full leaf node.

But what else do we have to do to maintain the structure of the search tree? We have to remove a splitter in their parent (the splitter separating these two leaves). What sort of problem could this cause? Well, the parent could have exactly $B/2$ items. If this makes the parent too empty, then we're going to need to recurse — we look at the sibling of the parent, and do the same sort of redistribute-or-merge. If we merge the parent with its sibling, then we need to delete a sibling from the grandparent, and so on.

If we get all the way to the root, we may actually delete the last splitter from the root. In that case, we remove the root and make its sole child the root. And that's how the tree gets shallower as we delete items.

So now we have our operations; let's talk about runtimes. We maintain a search tree with degree at least $B/2$, so the depth is $\log_B N$, which means search is optimal ($\log_B N$).

How much work do we do during an insertion or deletion? Let's take it one level at a time. On an insertion, you only touch $O(1)$ things per level — there's a path we traversed through nodes to do an insertion. Now we reach a leaf. If we're lucky we only update the leaf, but we might have to split the leaf, which means we might be writing 2 blocks to external memory. That may cause us to insert a new splitter in the parent, and if the parent is full, we may split the parent and write two blocks there. And we might write two blocks at the grandparent level, and so on. So at each level we're reading and writing at most two blocks, which means the total number of blocks read and written is $O(\log_B N)$.

The same applies to deletion — we go down a path, each node on that path might get merged with its sibling, but that's only two blocks being read and written per level, so it's $O(\log_B N)$ blocks being read and written in total.

So we also have this $O(\log_B N)$ bound for insert and delete.

What's the obvious question, now that we've proved this $\log_B N$ bound?

> **Question 30.1.** Can we do better?

The answer is no! This is optimal in the comparison model, for a very straightforward reason. In the comparison model, if we're looking for one item in a collection of N items, we need $\log N$ bits of information to identify the target item — this is just like the lower bound for sorting that we're familiar with (you need $n \log n$ bits to identify a proper permutation, each comparison gives you only one bit, so you need $n \log n$ comparisons). How many bits of info do we get when we read a block? You get B items; but are we really learning B bits? What we learn about this query is where in this order of B items is my query item. You could say we're learning information about the items in the tree, but in a sense that's information I already had. The new information I'm learning is the position of the query in the node that I'm reading, which is $\log B$ bits of information. That means I must perform $(\log N)/(\log B) = \log_B N$ queries in order to find where the items land among the N.

> **Student Question.** *What if you're not restricted to the comparison model?*
>
> **Answer.** There are much better structures for keeping collections of integers. Also you can use hash tables in external memory if you just want equality lookups. So going beyond the comparison model gives you a lot more power.

§30.3 Sorting

That's it for search trees; let's tackle one more fundamental problem, which is sorting.

How do we sort in external memory? Well, we just made a data structure for searching, which gives an algorithm for sorting — you can sort things by inserting them into an order structure, and reading them

out in order. So we can do B-tree-sort, which takes $O(N \log_B N) = O(\frac{N \log N}{\log B})$ time. So that's what we want to beat.

Question 30.2. Can we do better?

Well, let's pull off an off-the-shelf sorting algorithm — let's talk about mergesort. Using mergesort in a standard computational model takes $O(n \log n)$ time, through a natural recurrence. How do things change when we think about running mergesort in external memory?

Mergesort is based on a recursive series of merges. So how long does it take to merge two sorted lists in external memory? We've got two sorted lists in external memory; how do we merge them? Merge is basically a scan operation — the way you merge two lists is scanning through them, repeatedly pulling off the smallest element and moving that scan pointer. That's easy to transfer to external memory — you read the first block of both lists, and start peeling off small items and putting them into an output block. Whenever the output block gets full, you write it out. Whenever an input block gets empty, you read the next block from that list. So over the course of executing the entire merge, you just scan through both input lists and append out everything in the output list. This means merging two lists costs $O(N/B)$ time.

So with that in mind, we can write the recurrence for mergesort — to mergesort a list of N items, you break it in half and recursively mergesort the two halves. And then you have two sorted lists, which you can merge in N/B time. This gives

$$T(N) = 2T(N/2) + O(N/B).$$

And once you get down to a list of size B, you can sort it in one timestep by just reading it in, sorting it, and writing it out; so the base case is a list of B items. Basically each layer of merging is going to cost you N/B over all the different sublists; and the number of layers you're going to need is the number of powers of 2 between N and B. So the solution to this recurrence is going to be

$$O\left(\frac{N \log(N/B)}{B}\right)$$

(you have $\log(N/B)$ layers of merging, and each costs you N/B time).

Until now, we've been ignoring how to take advantage of memory. This mergesort algorithm only uses 3 blocks in memory at a time — all the rest of memory is being ignored, and it's being wasted. One observation is that if we've got all this memory, instead of thinking of blocks of size B we could stop at lists of size M (we could read in the whole list, sort it all at once, and write it all out). That replaces the $\log(N/B)$ with $\log(N/M)$, which is a small improvement.

But is there a better way to use all this memory? With search, we argued the standard memory model of binary splitting was inefficient because we actually got B items of information every time we read a block. We claim that similarly for mergesort in external memory, splitting into two subproblems is inefficient, because it's not using all the memory we can use. So what turns out to be better is to do a merge of *many* lists in parallel, using all of your memory. If we have a memory of size M, it means we can hold M/B distinct blocks in memory. So it is better to do an M/B-way merge. That's going to decrease the number of layers of merging needed from $\log_2 N$ to $\log_{M/B} N$.

How do we merge M/B lists efficiently using external memory? You can read the head block from each list into fast memory. And then we can keep peeling off the smallest items over and over again, into an output block. When an input block empties, we read the next block.

So this is making effective use of all our memory — keeping all these heads in memory at once. This lets us merge M/B lists in a single merge pass, and all of this takes N/B time for N items. So this improves the recurrence to

$$T(N) = \frac{M}{B} T\left(\frac{N}{M/B}\right) + O(N/B) = O\left(\frac{N}{B} \log_{M/B}(N/B)\right).$$

(You can still replace the N/B in the log with N/MB, but that doesn't change the asymptotics — you're getting basically all the value out of having M in the base of the log.) This is substantially better than before.

And jumping to answer the obvious question, this is actually optimal! This is the best runtime you can achieve for a comparison-based sort in external memory. It's the same sort of information-theoretic algorithm where you ask how much information you learn about the sort when you read a new block into memory. You need $N \log N$ bits of information, and when you read a block into memory, you've just read B items and have M items in memory, so you learn where these B items are among the M you had in memory. And you can work with that in order to prove this is a tight bound.

§30.4 More on sorting and searching

The reason we won't prove this is to have time to see one cool enhancement of external memory algorithms, developed at MIT.

> **Remark 30.3.** Prof. Karger has emailed out feedback on project proposals; if we didn't get any, this means he didn't see our project proposal, and we should resend it.
>
> The last day of classes is December 11th, and that is when the projects are due, because he's not allowed to set a later due date. In order for our own benefit, on December 9th (Monday), we're going to have a required peer editing session here in class. That means we need to have our project not complete, but in close enough to complete form that someone else in the class can read it and give you meaningful feedback on it. When David introduced peer editing, he noticed it dramatically improved the quality of projects being submitted, mostly because of how often peer editors said 'I don't understand what you're talking about.' (If the peer editors don't understand, quite likely the graders don't either.) This is a for-credit exercise (to be here, provide your project, and grade someone else's). So this is the one required attendance for the semester.
>
> David will put out a poll over Thanksgiving break to see if there's interest in continuing to lecture on Monday and Wednesdays (not Fridays). Don't feel bad about saying no; David will not mind not lecturing.

A couple of more details about sorting and B-trees — there is an annoying mismatch between our results for sortng and B-trees. (We saw mergesort, but you can also make quicksort work optimally — the trick is to split on multiple pivots instead o fa single one, to increase branching factor on a step and decrease the number of layers in the recursion.) The reason we get these time bounds is that all in-memory operations are free. In particular, the peeling off of the smallest items requires some heap in reality, and that's going to create computation costs that were ignored. But that is dominated in practice by the memory access costs, so that's all we pay attention to.

In standard sequential algorithms, we have $n \log n$ time bounds for sorting and $\log n$ time bounds for searching, and they kind of match (you can use a search data structure to give you optimal sorting). So they're kind of dual, in that the sort time for n items is n times the search time per item.

With these data structures (B-tree vs. mergesort) you don't have this match — using a B-tree to sort is much slower than actually sorting. So it seems like there's a sense in which B-trees are not optimal. But we just said they are, so how do you get around this?

There's a data structure called a buffer tree, which supports $O(\frac{1}{B} \log_{M/B}(N/B))$ time insert, delete, and search. How can this be, given the lower bound that we just proved (when we showed B-trees are optimal)? This is the *througput* time — if you do many operations, this is the cost per operation. And there is a very high latency — so the buffer tree data structure is kind of weird, in the sense you tell it you want to do a search, and sometime long after, it gives you the result (after you've done many other insertions and

deletions, you eventually get back the answer to the search query you made). There are lots of applications where this is okay. And this is actually substantially less than a cost of 1 per item (it's basically $1/B$). So this is a very fast data structure, but you have to sacrifice on latency in order to achieve it.

§30.5 Cache-oblivious algorithms

To give us a taste (they were invented here by students who took this class, so we can feel some sense of connection to the work): the idea of cache-oblivious algorithms came from thinking how annoying it is to have to tune algorithms to the particular parameters of your computer. These algorithms are very sensitive to the values of M and B (that determines the size of your B-tree blocks, and how many sublists you should merge). And that means every time you transfer your system to a new computer, you have to retune it. This really involves tuning, because there are constant factors that matter (so you need to figure out whether to use twice as many or so on). So there were lots of papers on tuning these.

Even worse, we don't just have one external memory. We have a memory hierarchy, with many layers; and for each there's a perspective where this layer is fast and the layer below is slow. Which of these layers is actually the bottleneck for your algorithm? That may depend on the constant factors associated with the relative speeds of the fast and slow memories.

So it's a nuisance to think about all these parameters. The idea of cache-oblivious algorithms is to try to develop algorithms that match the optimal external memory bounds *without knowing or using M or B*. The time bounds are expressed in terms of M and B — we can't get around that. But can we at least develop algorithms that don't pay attention to them, and achieve these runtimes anyway?

We assume that we have a sensible online paging algorithm. So given this assumption, we're just going to pretend that we're working in regular memory (we do random access to whatever values in memory we want; if they're not actually in small fast memory, then the smart external paging algorithm will fetch them and provide the value to us). So that's the assumption.

§30.5.1 Scanning an array

Just in order to convince us that this is even possible, let's start with the easiest thing: how do we scan an array? Well, you just walk down the array, reading one item at a time. How is that going to work if you've got a paging algorithm? You'll ask for the first item; there'll be a miss, so it'll fetch that block into memory. But it'll keep that item in memory for a while, so you can keep reading from it. When you get to the end of the block, there'll be another miss, and you'll walk down that block. So under any sensible paging algorithm, each block will be fetched once, and you'll read all the items in that block. So scanning an array costs you $O(N/B)$ external memory reads, even if you don't know or care about B.

§30.5.2 Divide and conquer — a general approach

But that's an incredibly simple problem. What about more challenging problems? The general approach for cache-oblivious algorithms is divide and conquer. If you think about recursive divide and conquer algorithms, they take your problem, break it into smaller ones, and recursively solve those. This meshes very nicely with external memory algorithms — you keep breaking into smaller and smaller problems, and eventually you get a problem that fits in memory. Then your paging algoirthm will take over and fetch all that problem into memory, and you'll solve it instantaneously (because computation is free). So divide and conquer creates small problems that can be solved fast; you just need to make sure the way you combine small problems works well with your external memory algorithm.

§30.5.3 Matrix multiplication

Last time we saw an explicitly designed matrix multiplication algorithm (break everything into $\sqrt{M} \times \sqrt{M}$ blocks and do blockwise multiplication). But we can be oblivious about this if we just do a natural divide and conquer. We think of our $N \times N$ matrix multiplication as a collection of matrix multiplications on matrices of size $N/2 \times N/2$ (to get the upper quadrant of the output, we take two products of quadrants from the smaller ones). This says

$$T(N) = 8T(N/2) + \text{a bunch of matrix additions.}$$

And as with scanning, matrix addition done in the obvious way (scanning across rows of the matrix) is going to cost $O(N^2/B)$. So we get

$$T(N) = 8T(N/2) + O(N^2/B).$$

Naively, this would look like an N^3 algorithm (if you went all the way down to the base case). But we don't need to go down to the base case, because we're in external memory. The point is that if we think about the recursion tree, we have layers of recursion at powers of 2 (with $N \times N$ at the top, then $N/2 \times N/2$ with 8 subproblems, then 64 subproblems of size $N/4 \times N/4$, and so on — it keeps growing geometrically while problem size shrinks). But once we get down to a size where a matrix fits in memory, the multiplication time is just the time to read the matrix. So once we shrink to size $\sqrt{M} \times \sqrt{M}$, the multiplication time is just $O(M/B)$.

Now if you think about it, the layers are getting more expensive as we go down (we're getting problems faster than the problems are shrinking), so it's the bottom layer of the recursion that determines the overall runtime, when we get down to size $\sqrt{M}$.

We stop at the level where $N/2^i = \sqrt{M}$, which is level $i = \log N/\sqrt{M}$. And at that level, we're going to have 2^i subproblems, and the total work we do is the work of scanning all those subproblems (N^2/B), which comes out to

$$8^i \cdot N^2/B = \frac{N^3}{B\sqrt{M}}.$$

This is interestingly exactly the time bound we got by breaking the matrix into blocks of size $\sqrt{M} \times \sqrt{M}$; but this recursive algorithm discovers that breaking without needing to know what M is.

So you get this time bound without actually having to optimize for M, or for B — all you're doing is scanning in your virtual memory model.

Of course there are better divide-and-conquer algorithms for matrix multiplication (e.g., Strassen), and they too work in a cache-oblivious model and give you even better runtimes.

§30.5.4 Binary search trees

In a similar vein, we can do cache-oblivious BSTs. We carefully designed B-trees based on the idea you should use a B-ary branching factor. But we'll think of BSTs as another kind of divide and conquer problem. And we've conveniently already done this. With Van emde Boas priority queues, we said that the search problem for something of size N can be thought of as two search problems for things of size $\sqrt{N}$ — you basically want to branch into $\sqrt{N}$ subproblems that each have $\sqrt{N}$ items. Here in cache-oblivious external memory, this turns into a clever layout of the BST.

We take our BST, and we imagine splitting it at the layer where there are $\sqrt{N}$ items. We lay out the BST in external memory by first laying out the top of the binary search tree (recursively), and then laying out each of the subtrees of the binary search tree, in order, also recursively. By recursive we mean we take the $\sqrt{N}$-item top of the BST, and we lay out the top of the tree with $N^{1/4}$ items and then each of the $N^{1/4}$ subtrees, and so on.

Eventually, we get dwn to little trees that have B items in them. And that's the base case for this recursion — if we use this layout (called the van emde Boas layout), we can now just use our regular BST algorithm — walk down the BST doing binary comparison at nodes, until we reach a leaf. Why does this work well in external memory? Well, because we've carefully laid out the tree so that when we're walking down a path, we're fetching in little trees of B items that are in single blocks. And because we fetch a tree in a single block, we traverse this whole little tree of B items with one block read. And so this little tree of B items we're traversing with a single block read essentially covers $\log B$ layers of this BST. This means we only need to read $\log N/\log B$ of these little single-block trees on our way to a leaf. So we only perform $\log N/\log B = \log_B N$ block reads. Again, this matches the performance of a B-tree, without the data structure being aware of what B is.

That unfortunately is all we have time for. It turns out you can also make our linked list trick work for cache-oblivious algorithms (you can achieve N/B scan time), and there are also cache-oblivious sorting algorithms (with the same runtime — there's one called *funnel sort*).

Student Question. *Is it always possible to translate one of our algorithms from the beginning to a cache-oblivious one?*

Answer. There's no known generic transformation algorithm that takes an external memory algorithm and makes it cache-oblivious — you have to think newly about the cache-oblivious algorithm. (Of course if you come up with a cache-oblivious algorithm you can also use it in the traditional external memory model.)

§31 December 2, 2024

Today we'll talk about parallel algorithms. As with other topics, we only have time to get a taste of the core models and ideas and approaches of this area; one could spend an entire semester learning it.

§31.1 Parallel algorithms

Parallel algorithms is an interesting subarea of TCS. It's gone through a very interesting series of phases, that are sort of out of sync with practice. In the 1980s, there was a tremendous amount of work on the theory behind parallel algorithms, because everyone thought they were the future just around the corner. But at that time it turned out every time someone built a sophisticated parallel machine, it took so long for them to do so that by the time they had it ready, the sequential processors had evolved so much they were as fast. So there wasn't a lot of use of parallelism in practice. Eventually theoreticians noticed this and stopped spending so much time thinking about them. But now we've hit the end of Moor's law and it's not possible for sequential machines to keep speeding up and eliminating the benefits of research in parallel machines. Now multicore is everywhere; that's a small amount of parallelism. In machine learning we're seeing chips with thousands of cores on them. So parallel algorithms are becoming pertinent and getting lots of attention again.

There's a huge gap between how theoreticians prefer to think about parallel algorithms and how they work in practice, which we'll talk about.

§31.2 Models

When you talk about parallel computation as a theoretician, there are two models we like to think about. One is what you're doing with circuits made of wires and gates (AND and OR and NOT, although theoreticians also like to think about more complex things like parity gates and majority gates and other gates computing

other fundamental functions). When you have a circuit, electricity can run through the whole circuit in parallel. If you have a graph (a DAG), all the gates can be computing at the same time within this network of wires and gates. That means there's an opportunity for parallel computation.

When you make these circuits, there's two things you try to optimize. One is the number of gates (you want to keep the circuit from getting too complicated). The other is the *depth* of the circuit, i.e., the longest path from an input to an output. We're assuming a *synchronous* model of computation, where all the input data arrives at the input wires, it feeds into a first layer of gates that all produce their outputs at the same time, those outputs are fed as input to the next layer, and so on. So in a sense, the depth of the circuit represents the time (in computational cycles) for the output to be delivered.

That's the first model of parallel computation — where depth equals clock cycles in a synchronous circuit.

The other model is what's known as pRAM, where p stands for parallel. This is a computer, like the ones we're used to, computing over a random access memory. But now we assume there are multiple processors all operating at the same time — in one compute cycle, each processor can read something out of memory, or perform some computation (addition or a conditional test or so on), or write something to a particular location in memory. You get parallelism by running all these processors at once. So you have a global shared RAM and multiple processors, that each do one operation per cycle of your computer. Those operations can be read or write or compute.

This model is very unrealistic; it's difficult to build a machine with many processors all accessing a single shared memory (the wiring gets very complicated; you also tend to have memory bandwidth problems, where if a thousand processors want to read different locations in memory, this creates an external memory situation where the memory is very slow and can't deliver data to all the processors in the same cycle).

So often when you build a massively parallel machine, you give each processor its own local memory, which it can access quickly; it's much slower to read from another processor. This starts to be more like a distributed system, where you have to think about routing information from one computer to a second; and you lose the notion that you can just read any memory location you want in one cycle, which is very simple and elegant.

But despite the unrealistic nature of pRAM, it's really popular because it's simple and elegant and you can easily talk about how algorithms work. It's kind of overly parallel, so it's great for exploring the maximum possible parallelism that you can achieve. It's kind of like spherical cows; it's a massive simplification of what you can really do, but a good starting place for thinking about the most basic part of how you parallelize. Once you have a good pRAM algorithm, you can think about lots of questions about making it parallel on a given parallel architecture.

Interestingly, both models are identically powered — anything you can do with a circuit you can do with a pRAM, and vice versa. We'll see this in more detail once we have more definitions on the board.

§31.3 The circuit model

Now we can get more specific about these circuits. We've said there's gates and wires, but you can specify different ways in which they're connected. One important distinction is *bounded* vs. *unbounded* fan-in and fan-out, which really is just another way of talking about the degree of the gates — how many input wires can feed into a single gate, and how many output wires can feed out? From a theoretical perspective you might want to say you can use as many as you want; but there are practical considerations because you have to drive power through that wire, so if you have a huge fan-out this can make the circuit harder to build.

Theoreticians have defined complexity classes.

Definition 31.1. $\mathsf{AC}[k]$ is the class of problems that are solvable using a $\mathsf{poly}(n)$ number of unbounded fan-in and fan-out gates and depth $O(\log^k n)$, where n is the input size.

We're going to see that $\log n$ plays the sort of same role as a fundamental unit of time in parallel algorithms as n plays in sequential algorithms — in sequential algorithms, you usually can't do anything in sublinear time, and we've developed lots of machinery for doing things that take time polynomial in the input size. Similarly in parallel algorithms, it's difficult to do things in better than $\log n$ time, but there's lots of things you can do in $\log n$ time, and even more you can do in $\mathsf{polylog}(n)$ time. We'll see some reasons why $\log n$ is a natural term.

Definition 31.2. $\mathsf{NC}[k]$ is defined in the same way, but with bounded (i.e., $O(1)$) fan-in and fan-out.

So here you're only allowed to use a constant number of inputs and outputs for each gate. (N is an abbreviation for 'Nick's class' — there was a person studying space complexity who developed a complexity class talking about how much space it takes to solve various problems. He had a buddy Steve and named it Steve's class, so now we call it SC. Later while Steve was studying parallel algorithms and developing complexity classes for them, he named his class after Nick.)

So we have this slight difference between bounded and unbounded fan-in. But they're not all that different:

Fact 31.3 — We have $\mathsf{AC}[k] \subseteq \mathsf{NC}[k+1] \subseteq \mathsf{AC}[k+1]$.

So if you have a problem that can be solved with unbounded fan-in and depth $\log^k n$, you can solve the same problem with bounded fan-in and depth $\log^{k+1} n$. So as you increase k, these stack up; the difference between bounded and unbounded fan-in is just a single log.

Definition 31.4. We define $\mathsf{NC} = \bigcup_k \mathsf{NC}[k] = \bigcup_k \mathsf{AC}[k]$.

We should also point out that we can and do talk about $\mathsf{AC}[0]$; there are interesting circuits that have constant depth. But to compute over nontrivial input they can't have bounded fan-in, so $\mathsf{NC}[0]$ doesn't make sense (those circuits would have to be of constant size). But $\mathsf{AC}[0]$ and $\mathsf{NC}[1]$ can solve large problems.

To prove this inclusion, we need to show how to transform a circuit of depth $\log^k n$ with unbounded fan-in into one with $\log^{k+1} n$ and bounded fan-in. We can do that by transforming each gate of the unbounded fan-in circuit into a structure with bounded fan-in. This is pretty straightforward — suppose we want to take an **AND** of n values, but we're only allowed to use bounded fan-in and fan-out. What kind of construction can we use? We can just pair things up and so on, creating a binary tree — so we take each unbounded fan-in gate, and turn it into a binary tree of bounded fan-in gates. And of course, this binary tree will have depth $\log n$; so each single gate in an $\mathsf{AC}[k]$ circuit turns into n gates in the $\mathsf{NC}[k+1]$ circuit, which multiplies the number of gates by n, which is fine (it's still polynomial). It also multiplies the depth by $\log n$, which is fine (because we're talking about polynomials in $\log n$). This is the first taste of why $\log n$ is the appropriate unit for parallel algorithms — a binary tree can be used in so many ways as a parallel circuit (if you just need to aggregate n things, you can do that in $\log n$ layers with constant work at each of the nodes in the tree). So adding up n numbers can be done in $\log n$ time; taking the maximum of n numbers; and so on. That's the start of the reasoning for why $\log n$ is a good unit for thinking about parallel algorithms.

Another reason that $\log n$ is so natural — let's think about what happens with divide and conquer algorithms. These are one of our most common paradigms for solving algorithmic problems. Sequentially, you tend to end up with a recurrence that looks something like

$$T(n) = aT(n/b) + \mathsf{poly}$$

(where the poly is for doing the cleanup). This turns into a runtime which is n^{poly}, even without the cleanup — we get a polynomial just from the shape of the divide and conquer.

But what would the analogous divide and conquer look like for parallel algorithms? Here your divide and conquer step looks like

$$T(n) = T(n/b) + \mathsf{cleanup}.$$

And this gets down to a constant size after just $\log n$ layers of recursion. So you're going to get something like $\log n$ times your cleanup cost. This is another reason $\log n$ is the natural quantity to think about when thinking about parallel circuits and algorithms.

§31.4 An example — addition

We'll see one example of desinging a circuit for how to solve an algorithmic paralell problem (and then move to pRAM). We've already seen some examples — any primitive aggregation problem can be done by a binary tree. But let's talk about something more sophisticated — parallel addition.

Example 31.5

We have two n-bit numbers a and b, and we want to compute their sum $a + b = s$.

What's the naive algorithm for doing this, and what's its runtime? The naive algorithm is to add bit-by-bit and carry. This is known as a *ripple adder* (if you turn it into a circuit). And we can actually write down that circuit. We'll use what's known as an *adder* as a basic circuit; the adder takes in two inputs a_i and b_i, and outputs the corresponding sum-bit s_i and a carry bit c_i. Of course, that carry bit is fed into the next adder, together with a_{i+1} and b_{i+1}; this affects the next sum-bit s_{i+1} and the next carry-bit c_{i+1}.

Ine ach block is a constant number of **AND**, **OR**, and **NOT** gates — you have three input bits and you can write down the logic table for what sum-bit and carry-bit should be produced for those three input bits, and then use a constant number of gates to implement all of that.

So to add two n-bit numbers, we need n of these adders, each which has a constant number of gates. This means we're using $O(n)$ gates, which is optimal — if you have n inputs, they have to be received through n gates (at least, working in NC). So that's great. What's the downside of this circuit? It also has $O(n)$ depth — the path defining that depth is the path of these carry-bits. So it's going to take you n synchronous cycles in order to get the output bit out of this circuit. That's something we can definitely improve on. So this solution does *not* demonstrate that addition can be done in NC or AC, because it's not logarithmic depth; to do better, we're going to need to dramatically improve the depth.

The technique we'll use is caleld *carry lookahead*. The intuition is that if we look partway down the adder circuit, really the only thing that connects one side to the other side is this single carry bit. So if we can plan ahead for the possible values of this carry-bit in the middle, then we can run a computation on one side before the computation on the other side finishes. We'll have to leave a hole of some sort for the ultimate arrival of this carry-bit, but we don't have to wait on the outcome of the entire computation.

So we're going to preplan for the late arrival of the carry-bit c_{i-1}. If I'm given a_i and b_i, there are three possible cases for what c_i is:

(We're focusing entirely on the carry bits because if we know the carry bits, everything else is trivial — then each of the gates by itself can compute the appropriate s_i. So all we need to think about is how we compute the carry bits.)

Case 1 ($a_i = b_i$). In this case, it doesn't matter what c_{i-1} is — if $a_i = b_i = 0$, then even if $c_i = 1$, the total value for that adder is 1, so you don't get a carry. Conversely, if $a_i = b_i = 1$, then even if $c_{i-1} = 0$ you still have an output carry. So $c_i = a_i$ in this case.

If $a_i = b_i = 0$, we call that a *kill*, where we set $c_i = 0$. If $a_i = b_i = 1$, we call that a *generate*, where we set $c_i = 1$. We'll represent these by k and g.

Case 3 ($a_i \neq b_i$). Here one of them is 0 and the other is 1; and then $c_i = c_{i-1}$. We call this a *propagate*, denoted by p.

So what we can do is, we can go back to our collection of adders, and just based on the incoming a_i and b_i values, we can write down whether each gate acts to generate, propagate, or kill the carry bit that's coming

into it. This is a simplification of the problem — we're no longer thinking about the a_i and b_i, just about the effect of each gate on the carry bits that are moving through these adders.

Of course, the very lowest order gate is going to receive a 0 bit coming in; that's its input, which it's going to either kill, generate, or propagate.

In order to decide what value traverses a certain arc, what we need to do is compose all the generate-propagate-kill steps before it, and apply that composition to the original incoming 0.

So let's think about that composition step. We can make ourselves a little multiplication table, where we think about two letters adjacent to each other, and the effect of composing them.

If one of the adders causes a kill, and the next adder also causes a kill, their combined effect is a kill — if both are killing a bit, the overall result is to kill the carry. If the first adder does a propagate or generate, you still get a kill — the second adder has the last word. The same reasoning applies if the second adder is generating — then it doesn't matter what the first adder does, you still get a generation. Meanwhile, if the second gate propagates, then you have exactly the same effect as the first gate.

(Rows are x_{i+1}, and columns are x_i.)

	k	p	g
k	k	k	g
p	k	p	g
g	k	g	g

So we have a little 'algebra' — a weird new operation on adders that takes a pair of adders and produces another, from the perspective of the carry bit.

Let's call this operation a *circle* operation (just to name it). To figure out the carry bits, we need to compute these products — we'll think of the initial 0-bit as just the outpot of a k (so we don't have to work with numbers at all). So we define $y_0 = k$, and then we define $y_i = y_{i-1} \circ x_i$. So we're defining the overall effect of all the gates recursively, from left to right. We're essentially looking at each prefix of this whole line of adders, and looking at the product of each prefix.

So if I compute all of these, if I know the value of each operation up to a particular point — if I know y_i (which I get by taking the product of all the adders to its right), then I can immediately read off the appropraite carry bit coming out of this gate. Why? I know the first thing that came in was a 0, and I just apply that 0 through the product of all these gates in order to find the value of the carry that comes out at this point in the chain.

Once we know this, we know $c_i = y_i(0)$ (i.e., y_i applied to an input 0). So all we have to do is compute all these y_i's.

And computing all these y_i's is what's known as the *parallel prefix problem.* What we'd like is a circuit or algorithm that solves all of these y's in parallel, so that we can use all of them at once to read off all the c_i's in parallel (and then we can compute all the s_i's in parallel, and we have the answer to our addition problem).

So hwo efficiently can we compute the product of n values in parallel? Well, one way to think about this, a sort of lazy way, is that we can do things in parallel — so how about we just compute each of them separately? If I can compute one of them in parallel, then I can compute all of them in parallel by using n times as many gates.

So let's look at the hardest one — the product of all n values. How quickly can I compute that? You can do this by divide and conquer, in $\log n$ depth — any single y_i can be computed using $O(n)$ gates arranged in a depth-$O(\log n)$ binary tree, just like doing a sum of n things (we called this a product, but we could have called it a sum; it's really a weird operator that's neither, and we can do a composition of n values in

a binary tree). This is already a win — we have a $\log n$ depth circuit, and we use one for each of the y_i's, resulting in a $\log n$ depth overall.

We're using a polynomial number of gates (n^2) and $\log n$ depth to get all the y_i, from which we get all the c_i, from which we get all the s_i; so we can do addition in $\mathsf{NC}[1]$.

All that's left is to make this a little bit more efficient. You don't actually need to use n distinct trees. If I look at my binary tree that I'm using to compute the product of all the y_i, I use the entire binary tree to compute y_n (the product of all the leaves). But we claim that while we're computing y_n, we're also getting a bunch of other useful values. What other values that I want are being computed while I'm computing y_n? The right child of the top node is computing $y_{n/2}$, and its right child is $y_{n/4}$, and so on. So for the work of computing a single prefix value, I'm actually getting $\log n$ different prefix values.

There's a lot of other values I'd *like* to compute, and I claim I've got some useful starting points for computing them. Let's ask, for example, about how I might want to compute $y_4 = x_0 \cdots x_4$. The point is to compute this, we already have almost everything multiplied — we only need to multiply in one more value.

More generally, all teh values computed in this subtree need, as their starting point, the value that was computed to their right; I just need to take the value computed here, and send it to all the nodes that need it on their left. After I propagate things up, I get some starting values. Then anything here — I need to compute the prefix sums just in this subtree, and then I need to multiply each of them by the value that was in the right sibling of this subtree. So to compute a particular subtree, I need to compute it locally and then multiply in what's coming from its right neighbor, if it has one.

What this means is if I work with this single tree structure but think of an up-pass followed by a down-pass, I can compute all n prefix sums just using the n nodes of this tree.

So the point is that we're computing various subproblems simultaneously as we pass values up the tree — we basically send up to each node the product of all its children. And then we distribute each node's value to all the descendants of that node's left sibling. So we compute the values of each node's children in an up-pass, and then we send each node's value to the descendants of its left sibling in a down-pass. And then each leaf computes its needed value. The result of this is that I get all $O(n)$ prefix products, using n gates and $O(\log n)$ depth.

(Of course the tree no longer represents the shape of the circuits — we have to replicate nodes for the downwards pass, and we'll have a sort of upside-down tree on top of the rightside-up tree to represent the entire circuit.)

> **Student Question.** *Do you only distribute to the right children of your left sibling, because they'll then distribute to their left siblings?*
>
> **Answer.** Yes — you want to do the accumulation as you go.

(This is how you do hardware addition; it's probably taught in 6.004.)

This was to give us one example of circuit design for parallel algorithms. Working with circuits is a nuisance — it's harder to think about because you have to get the wiring right, and the computational power of each gate is limited. So with more complex problems like graph algorithms, it's more natural to work with pRAM, which we'll move to for the rest of today.

Remark 31.6. AC[0] is one of the few complexity classes we understand quite well — because it's incredibly weak, we have strong lower bounds. For example, you can't compute the parity of n bits using **AND** and **OR** gates.

This algorithm is summing two n-bit numbers, which is different from adding n bits; Prof. Karger would need to think about whether you can do this in AC[0].

Also, this binary tree approach works for any associative function (it's not specific to kill-propagate-generate).

§31.5 The PRAM model

As we've said, in PRAM we have these processors and they can do single compute steps, which are allowed to read and/or write from shared memory. What are the metrics for a PRAM algorithm? Obviously the number of processors you're using is one thing you care about; the other is the time it takes to do the computation with all these processors.

We have these reads and writes to shared memory. It's possible for more than one processor to read or write from the same memory location at the same time-step, so a question immediately arises: what do we do about conflicts? Certainly if we're talking about writes, there's a serious potential for conflicts — two processors might want to write two different values, so what happens? (If two processors want to read, there's only one correct answer, but there may be hardware problems with the hardware trying to serve answers to two processors. There's also issues if one processor reads and another writes — do you get the old or new value?)

Theoreticians create refinements of the model to address these in different ways. One is the decision of whether to even *allow* concurrent reads and writes to a single memory location. If you define your model to be *exclusive-read*, then you're not allowed to use an algorithm that allows two processors to read from the same memory location at the same time (similarly for *exclusive-write*). We use shorthand to talk about the three most common cases — CRCW (where you allow concurrent reads and writes), EREW (where you don't allow either), or CREW (where you allow concurrent reads but not writes).

If you have concurrent writes, you need to think about tiebreaking rules — what happens if two processors write different values to the same memory location? There are many choices. One is that you get an arbitrary value. Another is that you get some value deterined by priorities of different processors (the higher-priority wins). You might get garbage you have no control over. The fact you have to worry about this tiebreaking is one reason people don't like thinking about concurrent write; it makes the model more complicated.

Although these models are different, they're not *that* different.

Definition 31.7. We define $\mathsf{CRCW}[k]$ as problems we can solve with $n^{O(1)}$ CRCW processors in $\log^k n$ time, and $\mathsf{EREW}[k]$ as the same using EREW processors, then we have

$$\mathsf{EREW}[k] \subseteq \mathsf{CRCW}[k] \subseteq \mathsf{EREW}[k+1].$$

The first inclusion is obvious; the second says you can simulate a concurrent-read concurrent-write algorithm with an exclusive-read exclusive-write system, with just a $\log n$ slowdown. How do you do this? We take the same perspective as in circuits — we ask, how can you make a single concurrent read (or write) into a series of non-concurrent reads and writes? If we can simulate a single read or write, then we can simulate the entire algorithm.

The idea is to repeatedly duplicate the value. So we have a single memory location, and we have all these processors that want to read from this memory location. How can they do it? You can use a binary tree structure — we have two processors that take turns reading from the memory location. Now we have two

copies, so each of those copies can have two processors that take turns reading the value. Now I have four copies, and so on. By spreading through a binary tree structure, I can propagate it to n processors in $\log n$ timesteps.

I can run the same thing in reverse for exclusive writes — if n processors want to write to the same location, they walk their way up a binary tree where you take turns (first the left processor writes its value to the parent and then the right one does; now you have half as many things that need to be propagated up, and so on; so after $\log n$ of these reduction steps, you have a single value that gets written into the actual memory location).

> **Remark 31.8.** If we have an algorithm using $\mathsf{poly}(n)$ processors, we can just associate with each memory location a tree that's big enough for all the processors, so that the processors know which locations they're writing to; and the processors who do want to write will be responsible for carrying the memory up the tree, while the others are somewhere else. (This is very space-inefficient in practice, but this result is mostly just for theoretical interest anyways.)

So now that we can simulate every single concurrent read and write, we know there's a $\log n$ difference between them. But they do differ — it's known that $\mathsf{EREW}[k] \subsetneq \mathsf{CRCW}[k]$ — there are things you can do with concurrent read that you can't with exclusive read.

Example 31.9

Suppose we want to compute the **OR** of n bits.

I have n processors, each holding a bit; I want to store, in some memory location, the **OR** of all of those bits. In a CRCW machine (we'll assume we have an arbitrary or priority model, where some written value gets preserved; if garbage results, this won't work). We'll have any one of the processors responsible for initializing the output to 0 (in fact, we can have them all write 0, to save the problem of choosing a leader). And then everyone with a 1 tries to write 1.

But with EREW, we can prove a lower bound of $\log n$. This is not a complexity class so we won't do it in detail. But essentially, in an exclusive read or write model, you can think from an information-theoretic perspective — what does each processor know about the input at a given time, and what does each memory location know? When a processor reads from a memory location, it can at best learn everything the memory location knows. When it writes to a memory location, now that memory location knows exactly what the processor knew. So each read can pick up some information about what a processor knew in the previous round.

Initially, each processor only knows about one value of the input. In the next round, each processor can learn about one other processor, which means in two rounds each processor only knows about two values of the input. The round after that, each processor can only know about four values of the input. By induction, after k rounds, a processor can only know about 2^k input values; this means until you've run $\log n$ rounds, it's not possible for any processor to know the answer.

So CRCW is strictly better than EREW.

§31.6 Computing max

OR was pretty easy; let's have some fun talking about **MAX**. Now I have n processors each with a number, and I want to compute the maximum value. This is a strict generalization of **OR** — if I can compute **MAX**, then I can compute **OR**.

> **Claim 31.10 —** In EREW, the time is $\Theta(\log n)$.

The lower bound is because even **OR** takes $\log n$ time. TO get a construction, we do the same binary tree thing — we can think of max as a binary operator, so this is the same situation where we're taking a product of n things.

What about CRCW? This is the first place where there's actually some algorithmic cleverness. (Here we'll use a comparison-based algorithm, where all you can do is compare numbers and branch off of that to do something.)

The primitive thing we have is the comparison of a pair of numbers — that's obviously going to be the starting point. But we're talking about parallel here. So if we're comparing pairs of numbers, why don't we start by just comparing *all* the pairs? Imagine I write all my numbers down, and I do n^2 comparisons — and I write a 1 or 0 dependingon the outcome of the comparison.

Can you, by looking at this matrix of 1's and 0's, say which one is the maximum value? You'll have a row with all-1's (we ignore the diagonal). So the maximum is a row that consists entirely of 1's.

Can we get a parallel algorithm that quickly finds a row consisting entirely of 1's? We can basically use the parallel **AND** algorithm for each row (the same as with **OR**); the only row that's going to get a 1 is the row that represents the max.

So we compare all pairs, compute the **AND** of each row, and then do a CRCW **OR** to find the identity of the row with an **AND** of 1. Note that when we gave the **OR** algorithm we said to write 1, but we could do an **ARG-OR** where instead of writing a 1 we wrote the location of the 1, which would let us read off which location has the 1. Similarly, here if we write the identity of the row in the CRCW **OR** computation, we can read that off.

So now we have an $O(1)$ time CRCW **MAX** algorithm. What's its downside? It does take n^2 processors. This is not unusual — often if you're willing to be very spendthrift with your processors, you can get a really nice speedup with your algorithm. But it's very inefficient — here if you're running n^2 processors, you're doing n^2 work overall, when we know there are algorithms that only take n computational work. So this is definitely overkill.

We can improve things a little bit. We can do this in $O(\log \log n)$ time with $O(n)$ processors. We'll do this recursively — if I have k items to find the max of, I make k^2/n *blocks* of n/k items. (This means I have $k^2/n \cdot n/k = k$ items, just divided into blocks.) I run the quadratic-processor **MAX** on each block. The block has n/k items, so this takes $(n/k)^2$ processors per block; and I have k^2/n blocks, so this means overall we're using n processors to do this.

But after we've found the maximum of each block, now we have one potential max per block. So we've succeeded in transforming the k items we started with to k^2/n items, using n processors and $O(1)$ time. In other words, we've reduced by a k/n factor.

So now we have a recurrence

$$T(k) = 1 + T(k^2/n).$$

Written a different way, if we write $k = n/2^i$, then we get

$$T(n/2^i) = 1 + T(n/2^{2i}),$$

which means we're basically squaring the denominator in each round; and this means after $O(\log \log n)$ rounds, you get down to $k = 1$.

(In the first iteration, where $k = n$, you go from n to $n/2$ in the obvious way, so you start with $k = n/2$ rather than n.)

Here, you can again see a common tradeoff — we force ourselves to be efficient in processors, and that costs us something in runtime, but by being clever we're still doing a lot better than the EREW algorithm — I'm still taking advantage of the ability to write many values onto the same location.

Student Question. *What about CREW?*

Answer. It depends. The lower bound that we argued for **OR** should still work — even though you're allowing concurrent reads, you can't get enough information into a memory location fast enough to let anyone collect it effectively. Many processors may know the same thing, but that doesn't allow every processor to know a lot.

But there are some algorithms where just concurrent read is enough to get better performance.

Student Question. *What if you thought of the number of processors as much smaller (e.g., keeping a variable for it)?*

Answer. In general, the number of processors will be a slowly growing function of the input size, rather than polynomial (if it's constant, that's the same as 1 processor). But what's nice about starting with pRAM is if you have an efficient algorithm (in terms of total work) using a massive amount of processors, you can simulate it with fewer processors. So if you solve the problem in pRAM, you can use the same algorithm with fewer processors. This is only worth doing, though, if you've made an efficient algorithm — one where the total work is not large.

We've seen two models; they're actually computationally identical, and we'll see this next time (remind David if he doesn't start with it next time — it takes 5 minutes to show they can simulate each other). So in fact the pRAM model is also described using the NC complexity class.

Wednesday will be our last lecture, and we'll see some more sophisticated PRAM algorithms. The course evaluations are open, so you should make sure to fill them out.

§32 December 4, 2024

§32.1 Circuits vs. PRAM

Today we'll continue our tour of parallel algortihms. Last time we saw two models — circuits (which involve gates, and you're interested in keeping the number of gates small and the depth of the circuit small) and the PRAM model (where you have a bunch of general processors all accessing (at the same time) a shared global RAM, and you're interested in the number of processors and the time, the number of synchronous steps they use). With circuits, we looked at addition of two numbers, and with PRAM we didn't get very far, but we looked at an interesting algorithm for **MAX**.

Last time we asserted but didn't prove that these two models are equivalence. We defined NC as the class of things that can be computed with a polynomial number of gates and polylogarithmic depth. We also talked about good computation in the PRAM, where you're using a polynomial number of processors and a polylogarithmic time bound. THe point is that these are the same complexity class — anything you can do with a polynomial number of gates in polylogarithmic gates, you can also solve using a polynomial number of PRAM processors in polylogarithmic time, and vice versa. So it's the same NC class being implemented in two different models.

This equivalence is actually not that hard to show. One direction is easy — suppose I'm given a circuit with a polynomial number of gates and polylogarithmic depth, for solving a particular problem. Can you design a PRAM algorithm for solving that problem? What that's basically asking is, can you simulate the circuit using your PRAM? If you can efficiently simulate the circuit, then you can run any circuit algorithm using PRAM.

We do this layer by layer — the fact that it has polylogarithmic depth means you can write the circuit as a certain number of layers of gates, where each layer only reads input from earlier ones. If you just put a processor onto each gate to execute the work of that gate, then you start with layer 0 — inputs stored in

memory. Then you simulate layer 1 of the circuit; a gate here only reads data from layer 0, in other words it only reads input values. Then it performs an **AND** or **OR** or whatever, so you just have the processor on that gate do those reads or perform the requisite operations, and now you have all the values for layer 1. We're in NC (the constant fan-in model), so the processor at that gate can read all the values it needs and perform the computation in constant time; this means the whole layer can be simulated by a polynomial number of processors in constant time. So the total time for simulating the circuit is proportional to the depth.

So any circuit can be simulated by PRAM. Now suppose we instead have a PRAM algorithm. We want to make a similar argument that we can always build a circuit to do what that PRAM did. This is a little bit more technical, and we're going to wave our hands in a few places. One thing we'll argue is that if you think about a single processor executing a single instruction in the PRAM, well, we know that processor is actually a circuit, right? So we're going to wave our hands and say having one processor execute a single instruction can be implemented by a circuit with polylogarithmic depth (even if it's a multiply, we've shown you only need polylogarithmic depth). So whatever the processor does is something that can be emulated by a NC circuit. So we have that building block, for one step of one processor.

How do we go farther and emulate the entire algorithm? The starting insight is that this algorithm involves a polynomial number of processors running for a polylogarithmic number of steps, so we can think of executing each timestep as a layer. We first use circuits to do all the things the processors do in timestep 0; so we can assign some circuit to have all those do their thing and produce some output, which becomes a layer of new values (we can think of values coming out of the processor as wires) which can then be used as inputs for the next timestep, which is implemented by a new layer of circuitry. There's one thing that worried David the most about this sort of simulation — processors have random access to memory. But circuits ahre hardwired in terms of the values they're reading. So can we emulate random memory accesses the way we need to when a processor decides to read a value from memory? We can certainly imagine there are wires holding all the values in memory at a particular timestep (we're only using a polynomial amount of memory, so we can have a gate that's outputting each value in memory somewhere, because memory is also just circuitry). But now when a processor says, I look at the value in such-and-such register, and I read that locaiton in memory — how can we make a circuit for that?

You can again have a binary tree kind of structure — imagine you build a binary tree over all the memory. And you pass values up the binary tree. What does each node in the tree do? Well, it has a third input, which is the memory location that you want to read. And if one of the things that's coming up — if it has the desired memory location in its subtree, then it passes that desired memory value up. Other subtrees don't have that memory location, so they can pass up whatever garbage they want. The point is each node on the path from the memory location to the root is reading an additional input that tells it that it has the desired value, so it passes that up; so only the desired value climbs all the way up to the root, and so at the root you get that desired value. That's how you simulate random accessing of a memory location.

You can do a similar thing when a write happens — it's a write to an unknown memory location, but you have these binary trees to decide what is the value stored in a particular locaiton, and they can collect a **OR** of all the writes that could have happened.

This is handwavy, but the point was to draw our attention to the fact that we have to think about this, and a general idea of how you address it.

So simulating one processor step is easy; simulating one memory access is doable; and that's basically what you need (random access plus computation in a processor can be done using circuits, and you compose all those circuits together and get a circuit for emulating PRAM). That's why these models are equivalent, and we can just talk about problems being in NC without worrying about which model we're implementing them in.

This is a very complexity-theoretic perspective; in practice, you'd choose to implement on PRAM very differently than with a circuit. But from a complexity-theoretic perspective, you're not changing the overall time and work bounds.

§32.2 List ranking

With that out of the way, we'll continue talking about PRAM algorithms, because having random memory access makes it much easier to talk about algorithms.

We'll see another fundamental important trick that's done in PRAM. We already talked about parallel prefix when we were doing multiplication for circuits — that's a fundamental technique for parallel algorithms. We used it to take all the prefix products or prefix sums of an array, and it took only a $\log n$ depth circuit with $O(n)$ processors.

It should be pretty obvious that you can do parallel prefix in $O(\log n)$ time with n processors, basically by simulating that circuit — you sum some pairs, and then fours, and then eights, and so on on the way up; and then you use the power-of-2 sums to assemble all the prefix sums that you need.

But again, PRAMs have memory and random access. So let's talk about the linked list version of this — instead of having your data stored in an array, suppose you had it stored in a linked list looping all over memory, and you again wanted to compute the prefix sums of the values stored in this linked list.

Now, one concrete example of this is the *list-ranking* problem.

Example 32.1

You're given an n-item linked list. You want to find the *rank* of each item in the list (i.e., the number of items in front of each item in the list).

This can also be seen as taking a prefix sum with a 1 in each position — if we add 1 for each item in the list, then the sum of all those values in list items in front of me in the list will give me the number of items in front of me in the list, which is my rank.

So we'll focus on this problem, but the algorithm we'll see is not limited to 1's; you can put whatever values in the list and sum them up. It's also not limited to +; you can use whatever combination operation you want. So this is basically equivalent to parallel prefix, but with a linked list.

In order to have *any* chance of doing this efficiently, we assume that we start with one processor on each item in the linked list. If we're talking about a general global shared memory machine, who knows what the memory allocator did — the actual linked list entries might be anywhere in memory. If I start out by giving you the tail of the list, and tell you to do list ranking, you first have to find all the other elements. And if all you have is a pointer, you'll have to follow the pointers to find all the other elements; that takes you n time, so you're already dead.

So we're going to assume that whatever created this situation where you have this linked list you want to rank, you've managed to keep one processor on top of each list item.

How do we solve this list-ranking problem? Sequentially it's trivial — you just start accumulating stuff or adding. So the sequential algorithm is that you just add incrementally from one end of the list. But this is going to take you n time. So how do we do this quickly?

Well, let's assume that each list item x has a data-field $d(x)$ and a next-pointer $n(x)$. And what we're interested in is for the list item to finish with the sum of lal the data-fields in front of it, instead of just its own value.

It turns out that there's a very elegant and simple way to solve this problem in parallel. What we do is, in parallel, every single processor says its data value gets incremented by the data value of its next element — so we add the value just in front of us. We also *shortcut* — we jump our pointer over the element immediately after us. So we set

$$d(x) := d(n(x)) \quad \text{and} \quad n(x) := n(n(x)).$$

So if we started out with a situation $x \to y \to z$, now we have $x + y$, $y + z$, and $z + \bullet$, but we've also jumped the pointer (so $x + y$ now points to $z + \bullet$, instead of $y + z$). This does exactly what we want — we incorporated the next value, but also forgot about it by jumping over it. So we preserve as an invariant that we have the sum of everything we've crossed so far by following pointers.

Claim 32.2 — The sum of what I can reach is unchanged.

The point is that there are various nodes I can no longer reach after the pointer-jumping, but all their values have been incorporated.

You can see why this is called pointer-jumping — we jump each pointer over the next value.

How fast is this algorithm? Each step doubles the 'jump range' of your pointer, so $\log n$ steps suffice. Another way to look at it is when you do this pointer-jumping step, you're taking one linked list of n items and turning it into two linked lists of $n/2$ items, which are no longer attached to each other at all. Again since you're cutting the list in half each time-step, $\log n$ steps suffice.

This is pretty good, and you can actually implement it as an EREW algorithm — it's pretty easy to ensure only one processor reads any given location and writes any given location in a single timestep. And this is another one of these problems with a $\log n$ lower bound — certainly this is harder than doing **OR** or **AND**.

So this is optimal in time. However, this algorithm is still not quite optimal in the strongest sense. One of the ways to think about the performance of a parallel algorithm, which we brought up last time — in the real world you'll never have a polynomial number of processors, you'll have 256 processors and a giant problem with a parallel algorithm using n^3 processors. You can't run n^3 processors, but you can simulate a many-processor algorithm with fewer processors, by giving your actual processors the responsibility of doing the work of many of the virtual processors. At the extreme, a parallel algorithm can even be simulated with just one processor, doing the work of all those processors in sequence.

Suppose we do that for this list-ranking algorithm. How fast is that algorithm going to run on a single processor? It's going to take $n \log n$ time, because we have $\log n$ time-steps and n processors (all acting). So it's not doing as well as the obvious single-processor algorithm.

So 'optimal' is actually used in a very specific way for parallel algorithms:

Definition 32.3. We say a parallel algorithm is optimal if the total work is equal to the work of the best sequential algorithm.

If you can figure out an algorithm that does no more work than the best sequential algorithm but is fast, then it's going to be good no matter how many processors you actually have — it'll get slower, but it won't lose out ot the best algorithms you could run on fewer processors.

So is there a way for us to modify our list-ranking algorithm to make it optimal? This is actually a tricky question, so let's back up.

First, can we do optimal parallel prefix of an *array*? Parallel prefix on an array is easier than pointer-jumping, because we have all the items nicely laid out in linear order. So let's try to do that first.

What are we trying to achieve? We've already argued several times that we're not going to beat $\log n$ time. So if we want a work-optimal algorithm, we can't fix the *time*. But if we can do it with $n/\log n$ processors instead of n processors, then the product of processors times time would be n, so our algorithm could be simulated efficiently on a single processor.

With $n/\log n$ processors, we can solve parallel prefix on $n/\log n$ items (in the same way as before). So we just need to reduce our problem by a $\log n$ factor. So what we do is we make blocks of $\log n$ items. And we sum each block using one processor. And now we have $n/\log n$ block sums, so we can parallel prefix the block sums. And now, for a given block, this reduced parallel prefix gives the sum of all the blocks in front

of me, and I again use one processor to compute the (at most) $\log n$ terms that extend from the sum of all the blocks in front of me. So I build the sums internal to each block using the next block sum as a starting point.

What makes this work is that we know we're going to be spending $\log n$ time anyways, so we can afford to give $\log n$ items to each processor and let it do some work on those $\log n$ items to reduce the problem.

This looks great; can we extend this idea to pointer-jumping? In fact, something goes wrong. What did we assume for pointer-jumping? At the starting point, we said, 'assume we have a processor on each item.' If we're only going to use $n/\log n$ processors, we *can't* start with a processor on each item.

In fact, it's a bit ambiguous where the processors started off, so let's define the problem.

Assume that the n list-structs are in an array, but out of order — we'll focus our attention on the n memory locations that contain linked-list elements, and we'll pretend they're all together in an array of n items. That solves the problem of knowing where the linked list items are. Now that we know where they are, we can tell a processor to go to this one or that one. But how do I divvy up the work?

What goes wrong with our argument from before? We can certainly make blocks of $\log n$ linked list items, and if we want to, as a first step, we can compute the sum of values in a block. But that's not actually going to help us, is it? Well, because this is a linked list — the block ordering is meaningless. What we need is to follow pointers to get values to add up. And the problem is, because of this arbitrary ordering of the pointers, we kind of don't know which items to assign to which processors. We might have one processor with a collection of items, but their pointers might go to a different block which is hypothetically being handled by another processor. On top of that, we have a second problem — in parallel prefix, I was able to do $\log n$ work to turn a collection of $\log n$ values into a single sum value. But when I'm doing parallel prefix, what happened in one step of pointer-jumping, I didn't turn a single list of n items into a list of $n/2$, but rather *two* litss of $n/2$ items. So there were still n items to worry about. So I don't get any shrinkage that lets fewer processors finish the work quickly.

Solving list ranking is actually harder. It *can* be done efficiently (with $n/\log n$ processors), but it turns out you need randomization. The trick is that what you really want to do is make the list smaller, so your smaller number of processors can use the naive algorithm. This means some of the list elements have to be thrown out. And you use randomization to decide which elements to throw away — each element flips a coin to decide whether it wants to stay or go. And the elements that have lost the coin-flip, you pointer-jump over *those* elements, and you only work on the pointer-jumping problem with the elements that survive. This lets you reduce the number of linked list elements to $n/\log n$. But it's more complex; and we won't go through it because we'll instead see some more advanced algorithms using the PRAM model. (You can go read about it if you want to. You can get almost as good deterministically using a trick called deterministic coin-flipping.)

§32.3 Parallel search

Unsurprisingly, if we're going to look at an interesting algorithmic problem in PRAM, what's the first thing we should look at — what's the sort of canonical algorithms problem? Sorting! So let's talk about parallel sorting.

Before getting to sorting, let's take a moment to talk about search. We have n items in a sorted array, and we want to find the predecessor of a given item. We know how to do search in $\log n$ time with one processor. That's already pretty fast. Can we do it faster if we have a lot of processors? In EREW, we once again have a lower bound that gets in our way (similarly to with **OR**). But if we're allowed to do concurrent operations, we can actually do better.

Suppose we have n processors. We'll use one per array entry to check if it is the correct predecessor. What work do I have to do in order to check whether a given item is the predecessor of the query item? Say we

have a list $abcdefgh$; how do we check whether d is the predecessor of 7? We need $d \leq 7 < e$. So if we have one processor per element, then in two timesteps the processor that holds the predecessor element can know that it holds the predecessor element. Now, how do we finish? The processor who knows it's the predecessor stores that value in the output cell. So in three timesteps, I'm able to find the predecessor of an element in the array.

Now actually, this last step — only one processor is writing to the output value. In fact, I read my own value, which is exclusive-read; then in the next step I read the next value, which is also exclusive-read. So is this actually EREW? The problem is that you need a concurrent read of the query value. So this is actually a CREW algorithm (concurrent-read exclusive-write) that takes $O(1)$ using n processors.

Is this a work-optimal algorithm? Not even close — sequential algorithms take $\log n$ work, and the parallel one takes n work, which is exponentially worse. This is typical — if you throw an insane number of processors, you can do really well in terms of speed. But it gets harder if you want to use not too many processors.

Any thoughts on how we could be a little more efficient in terms of search — instead of n processors, could we get down to $\sqrt{n}$? Maybe we could do this in a divide-and-conquer way. Imagine making blocks of $\sqrt{n}$ items. If I could figure out which block the predecessor is in, then we could point our processors at that block and solve the problem quickly. Can I quickly figure out which block the item is in? Well, we need to just look at the block boundaries and ask, which block boundary is the predecessor? So by doing the fast algorithm on block boundaries (using $\sqrt{n}$ processors) I can identify the block I belong in. Then I have a block of $\sqrt{n}$ items, and I can again use this to find the predecessor in this block.

This generalizes — I can work with $n^{1/k}$ processors by breaking into k levels instead of two. Put another way, we can solve this in $O(k)$ time using $n^{1/k}$ processors, by using multiple layers of this division process.

§32.4 Parallel sorting

So that's search. Now we're going to move on to sorting.

How do we approach this? We've got all these nice sequential algorithms, so the obvious thing to do is to ask about parallelizing them. And we actually have some recursive divide-and-conquer algorithms that naturally parallelize. Why? The point is that the depth of the recursion is logarithmic in a typical divide-and-conquer — if we divide a problem in 2, then after $\log n$ layers of recursion, we've divided into problems of size 1. So there are only logarithmically many layers, which is a great starting point for getting polylogarithmic runtime. The hard part is parallelizing the 'divide' — the work at a *single* level of the recursion.

For example, two of our canonical sorting algorithms are mergesort and quicksort. And each of these is parallelizable in a pretty natural way. Mergesort starts with lists of length 1 and merges pairs, then merges pairs into fours, then fours into eights, and so on. So there's $\log n$ layers of merging to complete a mergesort. That's great; all that's hard is parallelizing one merge.

Similarly with quicksort, we do a partitioning step, where we pick a pivot and split the elements into greater and smaller subparts. Again, there's $\log n$ layers of partitioning, so the tricky part is to paralellize one. We're actually going to steer away from quicksort because it's randomized (we'll study it next year); that makes it kind of tricky to analyze in parallel. When you analyze sequential quicksort, you can say the division is usually pretty even, so we can be pretty confident all the subproblems are the right size. When you do quicksort in parallel, you get a bit more nervous about the sizes of the subproblems and how they play out over layers.

Instead we'll talk about mergesort, which is deterministic and lets you not worry about those things. We want to merge two (sorted) length-n lists quickly in order to implement a mergesort. How might we do that?

One thing you could od is find a common splitter between the two list (some value); the things bigger than that value in each list come after all the things smaller; and then you can recurse. We're actually going to

use that idea in a slightly different form to get a more efficient algorithm. But we'll start with a more naive approach that gives a decent sorting algorithm.

We've got these two sorted lists. An item is in one of those sorted lists, so we know exactly where it belongs relative to the items in that list. What we need to figure out about this item to know where it should go is its position in the other list — we know the position in its own list, so if we also know its position in the other list, then we know how many items are smaller than it in the combination, and then we know where it belongs in the output list.

So we know an item's position in its own list, i.e., how many items are smaller. I want to figure out the same in the other list. Now I know the total number of smaller items in the merged list, i.e., my position in the output. So we're all set — if I just figure out my position in the other list, then I know exactly where I belong in the output, and we're done.

And how do I figure out where I belong in the other list? We can do this with binary search.

To set this up, we put one processor per item. Then it takes me $\log n$ time to figure out this location for each of the items, and then I can produce the output.

Is this optimal? It turns out it's not. We've got n processors taking $\log n$ time, so that's $n \log n$ total work. But if I just wanted to merge two lists using a single processor, that only takes n work. So this is actually wasting $\log n$ work.

It's only a subroutine, but if we think about how this plays out over the entire merge, when I have lists of size k it'll take $\log k$ time to do the merge with k processors. (The mergesort divides up problems — if we think of each item having a processor, when we split into subproblems we can still have one processor with each item.) So the total time is going to be $\log 2$ to get the merges into pairs, then $\log 4$, then $\log 8$, and so on, all the way up to $\log n$ for the final merge step. And if you think about it, in the middle of this sum there's a $\sqrt{n}$, and the second half of these terms are all going to be bigger than $\log \sqrt{n}$. So we're going to have $\frac{1}{2} \log n$ terms that are greater than $\frac{1}{2} \log n$, which means the overall runtime of this merge algorithm is $\log^2 n$ (using n processors). And so this is not quite an optimal sorting algorithm, because it does $n \log^2 n$ work (while sequential mergesort does $n \log n$ work).

We've talked about this in terms of optimality. Implementers will often instead talk about this as the cost of parallelization — if you try to chart out for a given number of processors, the *throughput* that you can achieve in terms of useful processing per time, ideally you'd like it to be linear (doubling processors gives you twice as much work). But parallelization often costs you efficiency, so you instead see some decaying curve where you get less and less efficiency the more processors you throw at a problem. And that's what we're seeing here.

There's two things we're unhappy with here — there's the efficiency question, and also this $\log^2 n$. The only lower bound we've talked about algorithmically is a $\log n$; do we really need $\log^2 n$, or can we get $\log n$?

Let's suppose we throw cost concerns out of the window, and say we're willing to use lots of processors. Then can we get a faster algorithm? You can replace binary search with a parallel search — we can use parallel search to merge in $O(1)$ time. This does cost us n^2 processors instead of n, but it gives us sorting in $O(\log n)$ time using these n^2 processors.

> **Remark 32.4.** Another way to do sorting is to use n^2 processors to do all pairwise comparisons in parallel — I compare every item to every other item. If I then find out how many comparisons I won, I know where I belong. But to find how many comparisons I won, I need to sum the outcomes of the comparisons. So this is another way to get $O(\log n)$ time sort. But we're still stuck with that $\log n$.

Cole currently has the nicest *efficient* sorting algorithm, which focuses on an efficient merge. We're going to use this same idea of an all-pairs comparison, but with fewer than n^2 processors.

First, we can sort $\sqrt{n}$ items in $O(\log n)$ time using n processors — that's what we just discussed (the brute force approach). We're going to use that idea to improve our merging. And it's going to use the pivoting idea (splitting the cells to be merged) brought up before.

The idea we'll pick up from here is that nearby items in the input go nearby in the output. So we don't actually need to do careful work merging each pair of items; we just need to be careful merging some equally spaced items in the two lists, and that will give us a guide for merging all the other items.

So again, we have a brute-force algorithm with lots of processors for merging two lists quickly, but we want to use fewer processors. Cole gave an algorithm that can merge n items of a list A with m items of a list B in $O(\log \log n)$ time with $m + n$ processors. This is going to give us a faster and more efficient mergesort. But let's describe how we do this.

We choose $\sqrt{n}$ evenly spaced splitters in A. And now, we use $\sqrt{m}$ processors to find the location of each A-splitter in B in $O(1)$ time. We talked before in terms of search, of how if we use $n^{1/k}$ processors, it only takes $O(k)$ time to find the location of an element in a sorted list. So with $\sqrt{m}$ processors, it only takes us constant time to find the location of an A-splitter in the B-list. (This is $\sqrt{m}$ processors per item, i.e., per A-splitter.) So the total number of processors is $\sqrt{mn} \leq m + n$. So we have enough processors to do this.

Now, we split B at the locations of the A-splitters. Let's call these *blocks* of B. What we know is that there are only $\sqrt{n}$ items of A among each B-block in the output list. And we know which they are — we found these A-splitters, which gives us groups of $\sqrt{n}$ items in A, and we now split B, and that tells us, for each group of items in A (between two splitters), which items of B it lands among. So we took these evenly spaced elements of A and located them in B. They might not be evenly spaced in B, but that doesn't actually matter; in each of these B-blocks, each of them gets some arbitrary number of items of B, but $\sqrt{n}$ items of A. So the first $\sqrt{n}$ items of A belong with whichever items of B cover those; that's the start of our merged list. So this is our first subproblem. Then the next group of $\sqrt{n}$ items of A also go with some items of B, and that's another subproblem.

And now we can recurse. Remember it only took constant time to do this splitting into subproblems. So our recursion is

$$T(n) = 1 + T(\sqrt{n})$$

(where the constant 1 is for splitting into subproblems, and then we're solving a problem with $\sqrt{n}$ items from the first list). What's weird is we don't care how many items are in the subproblem from the second list — that's not a parameter of the recurrence. What makes it work is that we're still sending one processor with each element of B — so if a subproblem gets lots of elements from the second list, it also gets lots of processors to handle them. This has $\log \log n$ recursion depth, so we can merge in $\log \log n$ time instead of $\log n$ time.

If we plug that merging algorithm into mergesort, then Cole's parallel mergesort takes $O(\log n \log \log n)$ time using n processors. So it's off by a $\log \log n$ factor from the best achievable runtime, and also off by a $\log \log n$ factor in terms of the total work.

But in the same paper, Cole shows how to 'pipeline' the merge phases — you kind of start one before the previous one is finished — such that you get $O(\log n)$ time with n processors. So Cole gave this optimal mergesort algorithm. And if you look at the details of what it does, it's actually a concurrent-read exclusive-write algorithm — it needs concurrent reads just as we discussed for search, but it doesn't use concurrent writes. So that's a pretty good result.

Hopefully that gave us a taste of parallel algorithms. Prof. Karger always wants to cover connected components but there's never time; all the problems we've studied in this class have been studied in the context of paralell algorithms, but we have to take another class for that.