# 6.1220 — Design and Analysis of Algorithms

Class by Piotr Indyk, Srini Raghuraman, and Bruce Tidor

Notes by Sanjana Das

Fall 2022

Notes for the MIT class **6.1220** (Design and Analysis of Algorithms), taught by Piotr Indyk, Srini Raghuraman, and Bruce Tidor. All errors are my responsibility.

## Contents

**1 Introduction** **6**

**2 Interval Scheduling** **6**
- 2.1 Unweighted Interval Scheduling . . . 6
- 2.2 Weighted Interval Scheduling . . . 7
- 2.3 Job Interval Scheduling . . . 8

**3 Lecture 2 — Divide and Conquer** **8**
- 3.1 Analyzing Runtime . . . 9
- 3.2 Order Statistics . . . 9
- 3.3 Integer Multiplication . . . 12

**4 Randomized Algorithms** **13**
- 4.1 What are Randomized Algorithms? . . . 13
- 4.2 A Toy Example . . . 14
- 4.3 Taxonomy . . . 14
- 4.4 Matrix Product Checker . . . 14
- 4.5 Quicksort . . . 16

**5 Amortized Analysis** **18**
- 5.1 What Is Amortization? . . . 18
- 5.2 Union Find . . . 18
  - 5.2.1 A Doubly-Linked List . . . 19
  - 5.2.2 Making FindSet Cheaper . . . 19
- 5.3 Amortized Analysis . . . 21
- 5.4 Improving UnionFind . . . 21

**6 Competitive Analysis** **23**
- 6.1 The Problem — Self-Organizing Lists . . . 23
- 6.2 A Practical Algorithm . . . 24
- 6.3 Competitive Analysis . . . 25
- 6.4 A Few Thoughts . . . 27

**7 September 27, 2022 — Hashing** **27**
7.1 Hashing Recap . . . 27
7.1.1 The Dictionary Problem . . . 27
7.1.2 The Simple Uniform Hashing Assumption . . . 28
7.2 Universal Hashing . . . 29
7.2.1 Constructing a Universal Family . . . 30
7.3 Preview of Perfect Hashing . . . 32

**8 September 29, 2022 — More on Hashing** **33**
8.1 Recap of Universal Hashing . . . 33
8.2 Perfect Hashing . . . 33
8.2.1 The Second Level . . . 34
8.2.2 Analysis of Storage . . . 35
8.2.3 Summary . . . 37
8.3 Open Addressing . . . 37
8.3.1 The Trouble with Chaining . . . 37
8.3.2 The Main Idea . . . 38
8.3.3 Search . . . 38
8.3.4 Deletion . . . 39
8.3.5 Efficiency . . . 39

**9 October 4, 2022 — Greedy Algorithms: Minimum Spanning Trees** **40**
9.1 Greedy Algorithms . . . 40
9.2 Some Definitions . . . 40
9.3 The Cut Property . . . 41
9.4 Kruskal's Algorithm . . . 42
9.5 A Stronger Cut Property . . . 44
9.6 Prim's Algorithm . . . 44

**10 October 6, 2022 — Maximum Flow I** **46**
10.1 The Setup . . . 46
10.2 Net Flow . . . 47
10.3 Describing a Maximum Flow . . . 49
10.4 Maximum Flow and Minimum Cuts . . . 50
10.5 Ideas for an Algorithm . . . 50
10.6 Residual Networks . . . 51
10.7 Augmenting Paths . . . 52

**11 October 18, 2022 — Max Flow** **52**
11.1 Review of Setup . . . 52
11.1.1 Net Flow . . . 53
11.1.2 Cuts . . . 54
11.1.3 Residual Networks . . . 54
11.1.4 Augmenting Paths . . . 55
11.2 The Max-Flow Min-Cut Theorem . . . 55
11.3 Ford–Fulkerson Algorithm . . . 57
11.4 Integer Flow . . . 58
11.5 Edmonds–Karp Algorithm . . . 58
11.6 Applications . . . 59
11.6.1 Bipartite Graph Matching . . . 59

**12 October 20, 2022 — Linear Programming** **60**
12.1 Introduction — A Toy Problem . . . 60
12.2 Duality . . . 61
12.3 Standard Form . . . 63
12.4 Feasibility . . . 64
12.5 Optimality . . . 64
12.6 Visualize . . . 65
12.7 Duality . . . 65
12.8 Rabbit out of Hat . . . 66
12.9 Why Care about duality? . . . 66
12.10 Applications . . . 67

**13 October 25, 2022 — Game Theory** **67**
13.1 Introduction . . . 67
13.1.1 An Easier Problem . . . 68
13.1.2 Mixed Strategies . . . 69
13.2 Equilibrium . . . 69
13.2.1 Dominant Strategy . . . 69
13.2.2 Equilibrium . . . 69
13.3 Normal Form . . . 70
13.3.1 Normal Form Games . . . 70
13.3.2 Best Responses . . . 71
13.3.3 Dominant Strategies . . . 71
13.4 Nash Equilibria . . . 72
13.4.1 Mixed Nash Equilibriums . . . 72
13.5 Zero-Sum Games . . . 73
13.5.1 Setup . . . 73

**14 Random Walks** **75**
14.1 Intro . . . 75
14.1.1 Time . . . 76
14.2 Markov Chains . . . 77
14.2.1 Stochastic Process . . . 77
14.2.2 Markov Chains . . . 77
14.3 Random Walks . . . 78
14.4 Back To Our Problem . . . 80
14.5 Mixing Time . . . 81

**15 November 1, 2022 — Random Walks II** **81**
15.1 Review of Last Lecture . . . 81
15.2 Expected Time to End . . . 82
15.3 Mixing Time . . . 83
15.3.1 Coupling . . . 83
15.4 Other Characteristic Times . . . 85
15.5 Markov Chain Monte Carlo algorithms . . . 85
15.6 Metropolis–Hasting Algorithm . . . 86
15.6.1 Choosing g . . . 87
15.7 Reversibility . . . 88

**16 November 3, 2022 — Intractibility** **89**
16.1 Introduction . . . 89

16.2 What Makes a Problem Easy? . . . 89
16.3 Types of Problems . . . 90
16.4 Polynomial Time . . . 91
16.5 Non-Deterministic Polynomial Time . . . 92
16.6 Reductions . . . 93
16.7 NP-Completeness . . . 93
16.8 Cook's Theorem . . . 94

**17 November 8, 2022 — Intractability II: Reductions** **95**
17.1 NP–Completeness . . . 95
17.2 Reductions . . . 95
17.3 SAT . . . 96
17.4 3-SAT . . . 98
17.5 Vertex Cover . . . 99

**18 November 10, 2022 — Approximation Algorithms I** **102**
18.1 Why Approximation Algorithms? . . . 102
18.2 Some Formalism . . . 103
18.3 Vertex Cover . . . 103
18.4 Set Cover . . . 105
18.5 Partition . . . 107
18.6 Greedy Vertex Cover . . . 108

**19 November 15, 2022 — Approximation Algorithms II** **109**
19.1 Linear Programming Relaxation for Vertex Cover . . . 109
19.2 Travelling Salesman Problem . . . 111
19.3 Metric Travelling Salesman Problem . . . 111

**20 November 22, 2022 — Exponential Algorithms** **114**
20.1 Introduction . . . 114
20.2 Dealing with Hard Problems . . . 114
20.3 Meeting in the Middle . . . 115
20.3.1 Subset Sum . . . 115
20.4 Branch and Bound . . . 117
20.4.1 3-SAT . . . 117
20.5 Conclusions . . . 119

**21 Novembe 29, 2022 — Online Learning: Multiplicative Weights Algorithms** **120**

**22 December 1, 2022 — Sketching and Similarity Search** **125**
22.1 Similarity Search . . . 125
22.2 Baseline Solution . . . 127
22.3 Approximate Similarity Search . . . 127
22.4 Algorithm 1 . . . 128
22.4.1 Random Sampling . . . 129

**23 December 6, 2022** **131**
23.1 Introduction . . . 131
23.2 Sublinear Space Data Stream Model . . . 131
23.3 Simple Examples . . . 132
23.4 Majority Element . . . 132
23.4.1 Generalizations . . . 135

23.5 Random Sampling . . . 136
23.6 Approximate Diameter of a Set in a Metric Space . . . 136

**24 December 8, 2022 — Distributed Algorithms** **138**
24.1 Distributed Algorithms . . . 138
24.2 Synchronous Network Model . . . 138
24.3 Leader Election . . . 139
24.3.1 Impossibility . . . 140
24.3.2 The Importance of Identity . . . 140
24.4 Maximal Independent Set . . . 141

# §1 Introduction

This year, our textbook will be the fourth edition of *Algorithms*; information for how to access it electronically is on Canvas.

There are two prerequisites for this course — 6.006 and discrete math (6.042 or 18.200). If these are not fulfilled, email 6.1220-admin@mit.edu. Describe your background and they will give you feedback about whether you should take the course.

Problem sets are released on Thursdays and due on Wednesdays. You should start psets early. The warmups are due in a day, with an automatic one-day extension; they are worth a tiny amount of points but they are important, and serve as motivators to start psets early. You can submit up to 3 psets up to 2 days late. The two lowest pset scores are disregarded; the same is true for warmups.

We will be describing algorithms, so it's important to do it right. We use the Goldilocks approach — the solution should have all the important things, but not more than that. You are strongly discouraged from submitting code; you should write in English. This is because it's more readable, and when you write code you usually make minor mistakes. Also, you should check all the boxes — describe the algorithm, describe runtime, prove correctness unless it's immediately obvious (or at least give a sketch). But don't be too verbose. (The provided solutions are reasonable examples.) The clarity of the solution si very important.

This course is titled Design and Analysis of Algorithms. 6.006 is more about algorithmic literacy — the focus is on the algorithms themselves. On the other hand, 6.046 is more about the *design* — the art and craft of algorithms. (You can think of 6.006 as a paintbrush and 6.046 as the actual painting.) We will see techniques such as divide and conquer, randomization, amortized analysis, greedy approach, incremental improvement, linear programming, dynamic programming, continuous optimization, reductions, approximation algorithms. We will also cover computing *models* (or frameworks). In 6.006 we saw two things — data structures (where we access data using certain subroutines) and *offline algorithms* (the algorithm takes input, reads it, does something smart, and writes the answer). Many algorithms in this course will be of this form, but we'll also see a few different frameworks:

- online algorithms (which don't see the full input at the beginning, and instead see it piece-by-piece and make decisions every time a new part of the input arrives — this is common in machine learning);
- streaming algorithms (which can read the full input but don't have enough space to store it);
- sketching algorithms;
- parallel and distributed algorithms.

# §2 Interval Scheduling

Today we will focus on one particular class of problems, and we'll see three different variants. This will give us a guided tour of various techniques we'll see later in the class. Interestingly enough, we'll see that similar-looking problems can have very different solutions and complexity.

## §2.1 Unweighted Interval Scheduling

**Problem 2.1.** THe input is a bunch of tasks. Each task has a starting point and an ending point (for example, we can think of the start and end time of a bunch of courses, like 6.046, 6.003, 6.036).

So we have $n$ requests modeled as intervals $r_i = [a_i, b_i]$.

You want to select as many intervals as you can. But you can't be in two places at the same time. So we say that two intervals $r_i$ and $r_j$ are compatible if they do not intersect, and our goal is to select the maximum number of compatible intervals.

**Example 2.2**

If we have *abacbc* with 6.046, 6.003, 6.036, the answer is 2.

If there are ties, we can select any one of the optimal ones.

Let's try to use a greedy approach. The greedy framework means we use a "simple rule" to pick the next $r_i = [a_i, b_i]$, add it in to the solution and discard all intervals that are incompatible, and repeat. This is called greedy because the way you select your interval is based on a simple rule — we only take the next interval into account (we don't worry about the future, only which interval to pick next).

The main part of the framework is what rule we use — of course some rules will work and some don't. For example, picking the interval that starts first doesn't work (for example, you could have *abbccdda*).

Our rule will be to pick the interval that *finishes* first. The intuition here is that such a interval leaves the most options for the future — we have as much space as possible to add future events.

**Theorem 2.3**

In each step, we select the interval that *finishes* first, meaning with the smallest $b_i$. The greedy algorithm with this rule returns the largest possible solution.

*Proof.* We use the "exchangeability argument" — let $r_1$, $r_2$, ... be the intervals selected by greedy (in this order), and let $r'_1$, $r'_2$, ... be an optimal solution (sorted in order of increasing endpoint). The idea is to step-by-step move the optimal solution to our solution, while keeping the cardinality the same. So our solution has at least the cardinality of the optimal solution.

Suppose $i$ is the first place where $r_i \neq r'_i$. By the selection rule, we know that $b_i \leq b'_i$. So then we can remove $r'_i$ from OPT and replace it with $r_i$, and this is still consistent with everything that follows.

So now we can take our solution ALG and step-by-step move our intervals to OPT, and at some point we've replaced everything. So each interval in ALG has a corresponding interval in OPT, which means our solution is at least as good as OPT. □

Now we want to find the runtime.

We first need to sort the intervals in ascending order of $b_i$. Then we consider each interval $r_i$ in order, remembering the last selected $r_j$. If $r_i$ is compatible with $r_j$, then we select it. Sorting takes $O(n \log n)$, but if you know the start times and end times are small integers (classes do not start at 11.0257, they start at 11) then you can do better with radix sort. So then you have the time to sort plus $O(n)$.

## §2.2 Weighted Interval Scheduling

Now suppose some of the intervals are more important than others — each interval $r_i = [a_i, b_i]$ has a weight $w_i$, and we want to maximize the total weight.

Solving this with greedy is surprisingly tricky (Prof. Indyk doesn't know a greedy solution). Instead we will use *dynamic programming* — which is essentially a recurrence with few subproblems. (We want the number of subproblems to be polynomial rather than exponential, so that you can tabulate the answers in polynomial time.)

Sort intervals by increasing start time. Then we want an recursive formula $\text{OPT}(r_1, \ldots, r_n)$. We can do this by focusing on one particular piece of the input $r_1$, and asking whether we should include it or not. This gives us two cases — we get one answer assuming it's included, and one assuming it's not. If we don't

include it, then the answer is just $\text{OPT}(r_2, \ldots, r_n)$. If we do include it, then the answer is $w_1$ plus OPT for all the intervals compatible with $r_1$, which are $r_t$, $\ldots$, $r_n$ where $r_k$ is the first interval with $a_k > b_1$.

So then

$$\text{OPT}(r_1, \ldots, r_n) = \max(\text{OPT}(r_2, \ldots, r_n), w_1 + \text{OPT}(r_t, \ldots, r_n)).$$

Now we can go through the mechanics. All the subproblems are for the suffixes of $r_1$, $\ldots$, $r_n$, so there are $n$ subproblems. We start from the shortest subproblems and expand.

We need to find $r_t$. A linear scan would take $O(n)$, while binary search would take $O(\log n)$, So the overall time is $O(n \log n)$.

### §2.3 Job Interval Scheduling

Now each job consists of $k$ *possible* intervals. We want to select the largest ste of nonconflicting intervals, where we select at most one per job.

This is clearly more complicated than the previous ones. In fact, this problem is NP-complete! (This means it's highly unlikely it has a polynomial time solution.) But a greedy algirithm finds a 2-approximation, and a better algorithm yields an $e/(e-1)$-approximation.

## §3 Lecture 2 — Divide and Conquer

Today's lecture is on divide and conquer. We've already seen this technique in 6.006, but it's a very important technique, so we'll reiterate it here. The types of algorithms we'll see here may be more intricate or elaborate than the ones we saw in 6.006.

We'll see two examples of divide and conquer:

- Finding the median in linear time.
- Multiplying two integers (more efficiently than the algorithm seen in grade school). This is important in cryptography — cryptography involves very large numbers, so we need fast ways of doing arithmetic with them.

**Algorithm 3.1** (Divide and Conquer) — The main idea of divide and conquer is the following:

1. *Divide* — we divide the problem into $a$ subproblems of sizes $n_1$, $\ldots$, $n_a$.
2. *Conquer* — solve the sub-problems recursivelly.
3. *Combine* — combine the solutions to the sub-problems into a solution to the original.

Some examples of divide and conquer that we've seen in 6.006 are:

- Merge sort — suppose we want to sort an array. Then we divide the array into two halves, and sort each of the two halves; then we combine these two sorted halves using the merge subroutine (where we have a pointer in each, and we keep picking the next correct element). Here we have 2 subproblems with size $n/2$.
- Binary search — suppose we want to find an element $x$ in a sorted array. Then we compare $x$ to the middle of the array. If $x$ is smaller than the middle, we recurse on the left, and if $x$ is larger, we recurse on the right. Here we have 1 subproblem with size $n/2$.

## §3.1 Analyzing Runtime

One aspect of divide and conquer algorithms is that they're relatively easy to analyze in terms of runtime — we have

$$T(n) = T(n_1) + T(n_2) + \cdots + T(n_a) + T_{\text{divide}} + T_{\text{combine}}.$$

Often all subproblems have the same size, giving a recurrence of the form

$$T(n) = aT(n/b) + T_{\text{divide}} + T_{\text{combine}}.$$

This is fairly straightforward to solve — the Master theorem applies directly to this setting and lets you solve this recurrence almost immediately. (There are different cases depending on how $a$ compares to $b$, but once you can write your runtime in this formulation, the rest is easy.)

## §3.2 Order Statistics

**Question 3.2.** Given a set $S$ of size $n$, how can we find the $i$th smallest element of $S$?

The answer would be the element in the $i$th cell if we sorted the array. But we don't actually have to sort the array to solve the problem — we only care about this one element.

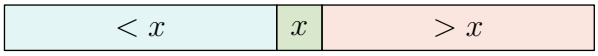
This is a natural generalization of a bunch of key questions — if $i = 1$ then the question is how to find the minimum, if $i = n$ then it finds the maximum, and if $i$ is around $n/2$ then it finds the median.

We already know how to find the minimum in linear time — we just perform a linear scan. The same works for the maximum. One way to find the median (or the $i$th element in general) is to just sort the array and find the $i$th element, which takes $O(n \log n)$.

To some extent, the sorting approach is quite wasteful — we really just want *one* number, but sorting gives us *all* of them, so we're computing lots of stuff and throwing out most of them. So we would like a faster algorithm — in fact, we'll see that we can do it in $O(n)$ time. For simplicity assume all elements of $S$ are okay.

First let's describe the general idea:

- Pick an element $x \in S$ "cleverly"; we call this the *pivot*.
- Consider the sets $L = \{y \in S \mid y < x\}$ and $G = \{y \in S \mid y > x\}$.




- We know $\text{rank}(x) = |L| + 1$. So we can perform the following case analysis:
  - If $i = |L| + 1$, then we are done.
  - If $i < |L| + 1$, then we recurse on $L$ and find its $i$th smallest element.
  - If $i > |L| + 1$, then we recurse on $G$. We know there's already $|L| + 1$ elements to the left of $G$ (which aren't in $G$), so we want to find the $(i - |L| - 1)$th smallest element of $|G|$.

**Student Question.** *Did we have to sort the array to get $L$ and $G$?*

**Answer.** No — we only rearrange the array so that the elements on the left are less than $x$ and the elements on the right are greater, but we don't sort these two halves. This allows us to only recurse on one of the two halves (if we sorted, we'd have to recurse on both of them).

**Student Question.** *How do you rearrange the elements of the array in linear time?*

**Answer.** If you don't care about space, you can simply create an additional array and add elements to the left and right. There *is* an algorithm which does this in-place (which is much more space-efficient) and still takes linear time.

We still have to pick the pivot, but before we explain our choice, we'll see how to make a good choice.

The runtime depends on the sizes of $L$ and $G$. Let's suppose we can pick our pivot in $O(n)$. Then if $|L|$ and $|G|$ are both at most $n/b$ for some $b > 1$, then we get

$$T(n) = T(n/b) + O(n),$$

and solving this recurrence gives $T(n) = O(n)$. (We can plug into the master theorem, but it's quite intuitive — we get the sum $n + \frac{n}{b} + \frac{n}{b^2} + \cdots$, which is a geometric series.) So if both sides are not too small, then we're in good shape.

But if the splits are very unbalanced, that's bad. In the extreme case, if we always have $|L| = 1$ and we always recurse on $G$, then we get

$$T(n) = T(n-2) + O(n).$$

This is very bad — it gives runtime $T(n) = O(n^2)$. (Intuitively, we get $n + (n-2) + (n-4) + \cdots = O(n^2)$.)

**Question 3.3.** How should we choose $x$?

The simplest way of choosing a pivot which *typically* leads to a balanced partition is to take one at random. This was the first linear-time algorithm for the problem, but the disadvantage is that the runtime is a random variable — in *expectation* it's $O(n)$, but we could potentially be unlucky and end up with $O(n^2)$.

Finding a pivot deterministically which gives a balanced partition is quite nontrivial. An ideal pivot would be the median, but of course this is a chicken-and-egg problem — our problem *is* to compute the median.

Now we'll see the full algorithm. (We call the algorithm Select.)

**Algorithm 3.4** (Blum–Floyd–Pratt–Rivest–Tarjan 1973) **—** First we pick the pivot:

1. Divide the $n$ elements into groups of 5. Find the median of each 5-element group by rote.
2. Recursively Select the median $x$ of the $\lfloor n/5 \rfloor$ group medians to be the pivot.

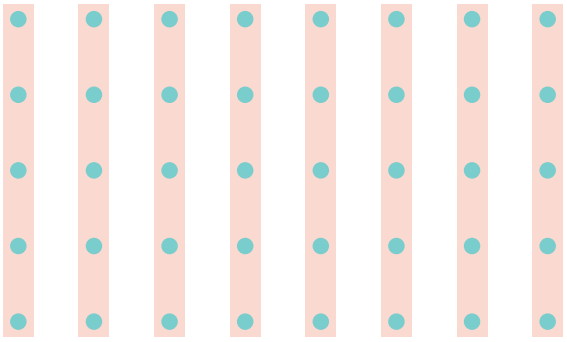
Now we perform the algorithm as described before:

3. Partition around $x$ into $L$ and $G$.
4. Then we have our three cases:
   - If $i = |L| + 1$, then we return $x$.
   - If $i < |L| + 1$, then we Select the $i$th smallest element in $L$.
   - If $i > |L| + 1$, then we Select the $(i - |L| - 1)$th smallest element in $G$.

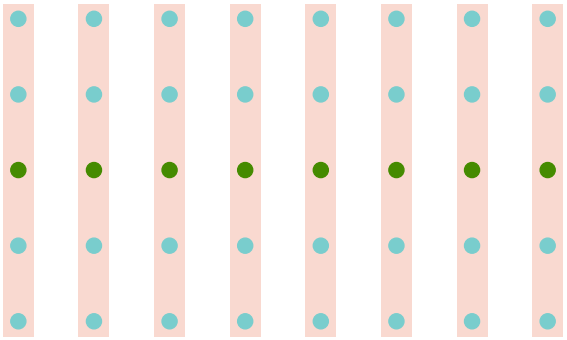
**Student Question.** *Is there anything special about* 5*, or can it be arbitrary?*

**Answer.** Larger numbers would also work, but it turns out 3 doesn't. We'll see why when we perform the analysis.

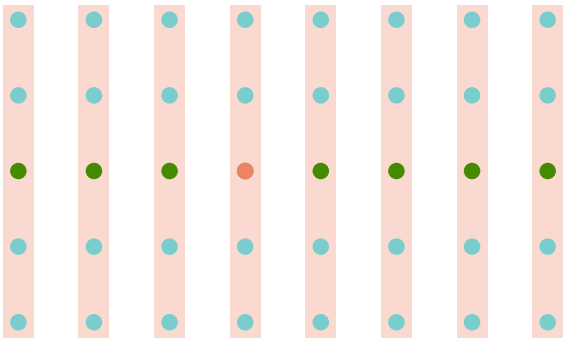
Let's see an illustration. First we divide the input into groups of size 5:



Now we find the median of each of these 5-element groups (this takes linear time):

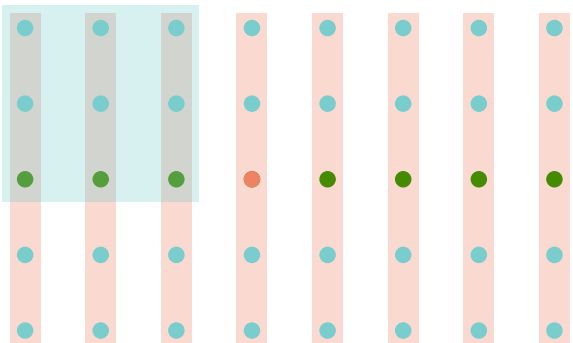


Now we find the median of these $\lfloor n/5 \rfloor$ group medians, using our algorithm recursively. (Our algorithm actually recurses *twice* — this means the recursive tree will quickly become weird. But it is probably best just to think about the whole recursive process just on the top level.)



Then $x$ doesn't have to be a true median, but it does give a pretty balanced partition: since $x$ is the median of medians, there are roughly $n/10$ medians smaller than $x$, and roughly $n/10$ medians greater than $x$.

But since these elements are themselves medians, each has at least 2 additional elements less than it (in its group). This means there's at least $3n/10$ elements which are at most $x$.



So then at least $\lfloor 3n/10 \rfloor$ elements are at most $x$, and similarly at least $\lfloor 3n/10 \rfloor$ elements are at least $x$ — so we get a pretty balanced split (at the cost of needing to use our algorithm again, and therefore expanding the recursion).

> **Student Question.** *It isn't generally true that $x$ is the true median, right?*
>
> **Answer.** Yes, $x$ is not necessarily the true median. But we're not returning $x$ — it's just a tool to make our algorithm more efficient. That's what makes the algorithm intricate — we have one recursive call to select the sub-median, but we have another to find the correct pivot.

So then our recursion involves at most $7n/10$ of the elements. This means our runtime involves:

- $\Theta(n)$ to divide the elements into groups of 5, and find the median of each 5-element group by rote.
- $T(n/5)$ to select the median of our $\lfloor n/5 \rfloor$ group medians to be our pivot, which we call $x$.
- $\Theta(n)$ to partition around the pivot $x$.
- $T(7n/10)$ to recurse on the correct half.

This means our recurrence is

$$T(n) = T(n/5) + T(7n/10) + \Theta(n).$$

If we didn't have our $T(n/5)$ term, then this would be easy — it's a standard application of the Master theorem. But unfortunately, we have this additional recursive step. So we'll instead use induction — choose some large $c$, and assume $T(m) \leq cm$ for all $m < n$. Then we get

$$T(n) \leq \frac{1}{5}cn + \frac{7}{10}cn + \Theta(n) = cn - \left(\frac{1}{10}cn - \Theta(n)\right).$$

As long as we choose $c$ large enough to handle the $\Theta(n)$ — meaning that $c$ is greater than 10 times the constant in $\Theta(n)$ — this is less than $cn$, and we are done by induction.

> **Remark 3.5.** There is the tiny issue that $n$ may not be divisible by 5. This is not hard — if $n < 5$ then the algorithm can't be run, but we can simply solve the problem by rote.

## §3.3 Integer Multiplication

> **Question 3.6.** Given two numbers $a$ and $b$, each with $n$ bits, how can we compute the product $ab$ (with at most $2n$ bits)?

In cryptography, it's not unusual to have thousands of bits. Here note that everything is in binary.

One way is to use the grade school algorithm — take the first number and multiply it by every single digit of the second, and add it up. This gives $O(n^2)$ time. But this is not great for cryptography. So we will see a faster way, which unsurprisingly uses divide and conquer.

First we need some way of dividing. To do this, we could split $a$ and $b$ in the middle — we write $a = 2^{n/2}x + y$ and $b = 2^{n/2}w + z$. Literally, $x$ is the first half of $a$, and $y$ is the second half of $a$, when written in binary.

We then have

$$ab = (2^{n/2}x + y) \cdot (2^{n/2}w + z) = 2^n xw + 2^{n/2}xz + 2^{n/2}yw + 2^{n/2}yz.$$

So we could get *some* divide and conquer algorithm by computing $xw$, $xz + yw$, and $yz$ — if we could compute these, then the rest would take $O(n)$ time.

If we implement this approach, then we make 4 multiplications of $n/2$-bit numbers. So we get

$$T(n) = 4T(n/2) + O(n).$$

By the Master theorem, this solves to $T(n) = O(n^{\log_2 4}) = O(n^2)$. We are not happy with this — we already know how to get a $n^2$ running time, from the grade school algorithm, so this isn't an improvement of anything we know.

In order to get a better algorithm, it would be enough to replace 4 with 3 — then we'd end up with an exponent of $\log_2 3 < 2$. So we'd like to somehow only use 3 multiplications instead of 4.

The key idea is that we do need to know $xw$ and $yz$, but we don't actually need $xz + yw$ — we only need their sum. So there is the hope that we can compute this sum without doing two recursive calls.

**Algorithm 3.7** (Karatsuba 1962) — First compute $xw$ and $yz$. Then compute $(x+y)(w+z)$. We have

$$(x+y)(w+z) = xw + xz + yw + yz,$$

and since we've already computed $xw$ and $yz$, we can calculate $xz + yw$ as

$$xz + yw = (x+y)(w+z) - xw - yz.$$

This isn't the most efficient algorithm, but it's the first one people found that's more efficient than the grade school one.

Note that $x + y$ and $w + z$ still have roughly $n/2$ bits, so all our recursive calls take $T(n/2)$. And we only have 3 multiplications! So then we get

$$T(n) = 3T(n/2) + O(n).$$

By the Master theorem, this solves to $T(n) = O(n^{\log_2 3})$, which is a faster algorithm!

# §4 Randomized Algorithms

Today we'll cover a technique that we haven't seen before (in 6.006) — randomization. A randomized algorithm is essentially an algorithm that flips coins. Two examples we'll see are a matrix product checker (as a toy example) and quicksort.

## §4.1 What are Randomized Algorithms?

**Definition 4.1.** A **randomized algorithm** is an algorithm whose execution is not deterministic given the input, but depends also on random choices made in various steps of the algorithm's execution.

There are various ways of formalizing this. One way is to assume that the algorithm has access to a random number generator — on every call to the random number generator, it outputs a fresh sample $x$ from $\{1, 2, \ldots, R\}$, which is independent from all previous random numbers. The algorithm's decisions should depend not only on the input but also on the random numbers being sampled.

Randomization is useful because it sometimes lets us do things we wouldn't otherwise be able to do — it increases functionality (and we'll see an example in a moment). In many other cases, we *can* make the right choice (we could select the right number, or the right procedure), but doing it would take lots of time and code. In situations like this, randomization lets us make our decision in a much simpler way. Quicksort will be an example of this.

**Student Question.** *What's the runtime of the random number generator?*

**Answer.** We assume it's $O(1)$ — we assume arithmetic operations, lookup, and random number generation all take $O(1)$. But in the actual world, this is a more complicated question — where random numbers come from is nontrivial.

## §4.2 A Toy Example

First, let's see why randomization can be useful.

> **Question 4.2.** You are given two cups, and there is a coin hiding in one of the two cups. You play against an adversary for $T$ rounds; in each turn, you collect the coin if you guess where it is.

If you have a fixed (i.e. deterministic) strategy, then the adversary can always do the opposite. For example, if your adversary knows that you always select the cup on the right, then the adversary will always put the coin on the left. So your total payoff is 0 — this is true for *any* deterministic strategy.

However, if we choose a *uniformly random* cup on every round, then no matter what the adversary does, you are right with probability $1/2$, and therefore your payoff is $T/2$!

So randomization makes it possible for the algorithm to fool the adversary, given enough choices — because no matter what the adversary does, they cannot know the random coins, so they cannot design an input to your strategy.

## §4.3 Taxonomy

There are two types of randomized algorithms:

- **Las Vegas** algorithms have always correct output and *typically good runtime* (but are sometimes slow).
- **Monte Carlo** algorithms have always good runtime and *typically correct output* (but sometimes the output is garbage).

Quicksort is of the first type — it'll always sort the array, but it could potentially take a long time. Meanwhile, Monte Carlo algorithms

> **Question 4.3.** Is one always better than the other?

In a way, it depends on what you're doing. But suppose that you are interested in this question from the asymptotic perspective.

Then you can use a Las Vegas algorithm to simulate a Monte Carlo algorithm — if you don't care about constants (everything up to a factor of 2 or 10 is the same), then if you know the expected runtime is something, the probability the runtime exceeds this bound by 10 is at most $\frac{1}{10}$. So you can simply terminate the algorithm after 10 times the runtime; this guarantees you get the correct output with probability $\frac{9}{10}$. Meanwhile, you can't usually go the other way around.

> **Student Question.** *When would you use Monte Carlo algorithms?*
>
> **Answer.** We can control the probability of success by repeating multiple times. For example, if the algorithm fails with probability $\frac{1}{2}$, that is not a great algorithm. But if you could be wrong with probability 1 in a billion, you could live with this. Monte Carlo algorithms are useful because they're simple and fast, so you can repeat them multiple times — and that makes the probability of failure go down exponentially. (For example, this is how you check a number is prime.)

Note that these properties hold for *any* input — "typically" is defined with respect to the random choices of the *algorithm*.

## §4.4 Matrix Product Checker

**Problem 4.4.** Given three $n \times n$ matrices $A$, $B$, and $C$, we want to check whether $AB = C$.

This is somewhat of a toy example, but its generalization is actually quite deep.

A deterministic algorithm would be to actually compute the product $AB$. The simple algorithm takes $O(n^3)$ time. There are faster algorithms, but they are not very simple — Strassen (1969) uses a similar idea to the one we've seen for number multiplication, where we multiply two $2 \times 2$ matrices using 7 multiplications (which gives $O(n^{2.81\cdots})$ runtime). In 1990 Copersmith–Winograd got it down to $O(n^{2.376\cdots})$, but this algorithm is horrible enough that probably no one has implemented it. The record-holders are Alman–Vassilevska Williams 2020, who got down to $O(n^{2.372\cdots})$. This is the record, but it is even more complicated. So in short, we do not know how to perform the multiplication much faster than $n^3$. It's still a huge open problem whether it can be done in $O(n^2)$.

We will not solve this problem, but we will give an $O(n^2)$ algorithm that:

- If the answer is yes, then our algorithm outputs yes with probability 1.
- If the answer is no, then our algorithm outputs yes with probabilitiy at most 0.001.

We'll actually make an algorithm with probability $\frac{1}{2}$, and then repeat the process to obtain one with probability 0.001.

Note that this is a Monte Carlo algorithm — it always runs in a given time, but may make errors.

**Algorithm 4.5 —** Choose a random binary vector $x = (x_1, \ldots, x_n)$ such that $\mathbb{P}(x_i = 1) = \mathbb{P}(x_i = 0) = \frac{1}{2}$ for each $i$.

Then output yes if $ABx = Cx$ and no otherwise.

One way of thinking about this is that $AB$ and $C$ are both programs — they take vectors $x$ and output $ABx$ and $Cx$. We're testing whether these programs give the same answer, so to test that, we generate inputs at random.

To make sure this runs in $O(n^2)$ time, instead of multiplying $AB$ first and then multiplying by $x$, we first calculate the vector $B(x)$, and then multiply this vector by $A$ to find $A(B(x))$. Both multiplications can be done in $n^2$ time.

**Remark 4.6.** This is a very powerful trick in matrix computation — $(AB)x$ would take $O(n^3)$ time to calculate, but we'd never do it because it's equal to $A(B(x))$.

**Remark 4.7.** There is a classic dynamic programming problem where you're given a sequence of matrices $A_1 \times A_2 \times \cdots \times A_n$ (with different but correct dimensions), and we want to figure out the optimal order to perform the operations to optimize runtime. Our observation here is a special case of that.

Now let's find the probability that it's correct. Let $D = AB$.

**Case 1** ($D = C$). Then of course $Dx = Cx$ for all vectors $x$, so we always get the output YES.

**Case 2** ($D \neq C$). Then we'd like to argue that there's a good probability we get different answers.

**Claim —** $\mathbb{P}(Dx = Cx) \leq \frac{1}{2}$.

*Proof.* First, there certainly *exists* $x$ such that $Dx \neq Cx$; we want to show that this is true for at least half of $x$.

Suppose there is some entry of $D$ and $C$ which differs. Let's simplify the problem; forget about everything in the matrices except the relevant rows, which we call $d$ and $c$. Then $\mathbb{P}(Dx \neq Cx) \geq \mathbb{P}(dx \neq cx)$.

First, we can rewrite $dx \neq cx$ as $(d-c)x \neq 0$. Since we assumed $d$ and $c$ differ at some position, which we call $i$, then we can write

$$(d-c)x = \sum_{j \neq i}(d_j - c_j)x_j + (d_i - c_i)x_i,$$

where $d_i - c_i \neq 0$ (but we don't really know anything about the other $d_j - c_j$).

Now if $x_i = 0$, then $(d-c)x = S_1$. Meanwhile, if $x_i = 1$, then $(d-c)x = S_2 \neq S_1$. (Intuitively, we can think about first generating everything except $x_i$, and then flipping $x_i$ — then we get one of two different numbers.) Then at least one of $S_1$ and $S_2$ is nonzero, so the probability that $(d-c)x \neq 0$ is at least $\frac{1}{2}$. (It could be that the probability is much greater, but that's fine.) □

**Remark 4.8.** We could reduce this probability further by having more choices for $x$ — for example, choosing $x$ from 0 to 10 rather than 0 to 1.

**Student Question.** *If you could choose $x$ from an infinite set, could you reduce the probability of failure to 0?*

**Answer.** Yes, if you had a real computer (which does operations with reals, and does not exist).

To conclude, this algorithm has the desired property — if $D = C$ it outputs YES, if $D \neq C$ it outputs YES with probability at most $\frac{1}{2}$. If we want to reduce the probability to $\frac{1}{4}$, we can run the algorithm twice and output YES if and only if *both* runs return YES. Then this occurs if and only if both runs are wrong, and since they're independent this has probability at most $\frac{1}{4}$. Similarly, we can reduce the probability down to $\frac{1}{8}$, or $\frac{1}{16}$, or 0.0001.

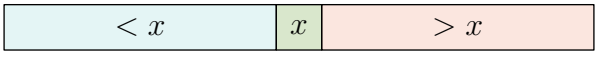
## §4.5 Quicksort

Quicksort is one of the oldest sorting algorithms. It was invented by C.A.R. Hoare in 1962. It's a divide-and-conquer algorithm that sorts "in place" — like insertion sort, unlike merge sort. It's very practical, and it can be viewed as a randomized Las Vegas algorithm. (It always sorts correctly, but its runtime can vary.)

**Algorithm 4.9 —** To sort an $n$-element array:

1. Choose a pivot $x$.
2. *Divide* — partition the array into two subarrays such that the elements in the lower subarray are at most $x$, and elements in the upper subarray at least $x$.
3. *Conquer* — recursively sort the two subarrays.

Note that there is no combine step —- once we sort the two subarrays, we're already done.




First let's analyze the worst case. If we always pick the min or max, then one side of the partition always has no elements. So we get $T(n) = T(0) + T(n-1) + cn$ (since partitioning takes $cn$ time), and therefore $T(n) = \Theta(n^2)$.

Note that this is why we don't want to pick the pivot as the *first* element — often, the input is already somewhat sorted, so the first element might be small.

Meanwhile, if we're very lucky, then we always split the array evenly, to get

$$T(n) = 2T(n/2) + \Theta(n),$$

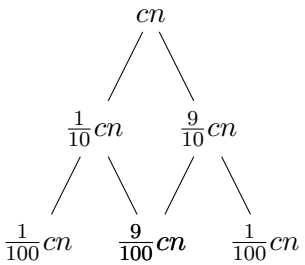
and therefore $T(n) = \Theta(n \log n)$.

We *could* simply use the median-finding algorithm from last class — we know there's a purely deterministic algorithm that computes the median in linear time. But then the algorithm would lose its simplicity, which is one of its main advantages — our code is only simple if our method of choosing the pivot is simple (every other step is simple). And the median-finding algorithm *is* linear, but it's not that fast — it's a relatively large constant.

To the surprise of no one, we pick the pivot at random. To see why this helps, suppose we're only *somewhat* lucky, and our split is $\frac{1}{10} : \frac{9}{10}$. Then our runtime recurrence is

$$T(n) = T(n/10) + T(9n/10) + \Theta(n).$$

This still solves to $n \log n$ — we can see this by falling back on a recurrence tree.




Note that at every level, our coefficients sum to 1 — since we're simply splitting up the array. So then we can upper-bound the work by multiplying $cn$ by the depth of the tree. So as long as the depth is at most $\log n$, the total work is $O(n \log n)$!

This makes sense, because in the worst-case scenario, the tree was very deep. So we just want logarithmic depth — and it's easy to see that in the somewhat-lucky case, the depth *is* logarithmic, since the length of the input decreases by a constant factor. Here the depth is $\log_{10/9} n$.

If the split is *no worse* than $\frac{1}{10} : \frac{9}{10}$, then we still get $O(n \log n)$ — the tree still has sum $cn$ at each level, and at most $\log_{10/9} n$ depth.

We will now choose randomly. We'll actually perform the algorithm ParanoidQuicksort — choose a pivot at random and perform the partition. Then if our split is unlucky, we keep repeating the process until we get a split we're happy with — one no worse than $\frac{1}{10} : \frac{9}{10}$.

Now let $T(n)$ be an upper bound on the *expected* runtime on any array of $n$ elements. Then

$$T(n) \leq \max_{\frac{n}{10} \leq i \leq \frac{9}{10}} ((T(i) + T(n-i)) + \mathbb{E}[\#\text{trials}] \cdot cn.$$

So it's enough to show that the expected number of trials is constant — in fact, we'll show it's at most $10/8$.

Every trial, the probability that a random pivot induces a somewhat lucky partition is $\frac{8}{10}$ — because if we imagine the array being already sorted, then we get a split that's lucky if the pivot is not too close to the left endpoint or the right endpoint (in particular, it cannot be within $\frac{1}{10}n$ of the left or right). But every other choice of pivot gives a somewhat lucky split, and that's $\frac{8}{10}$ of the array.

And if we flip a coin with an $8/10$ probability of success, then the expected number of trials to get a success is $10/8$.

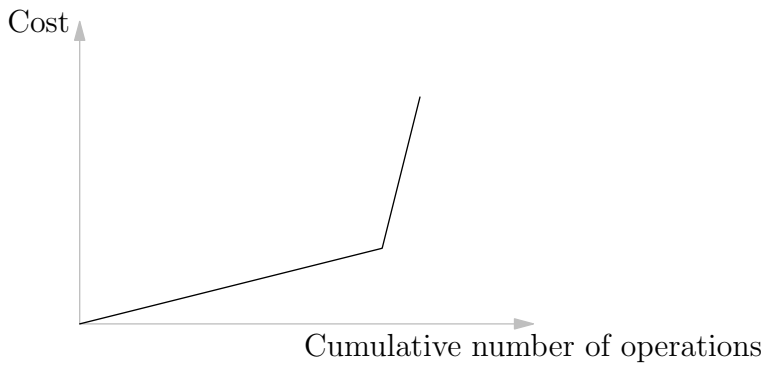
The full analysis of actual quicksort is more messy.

# §5 Amortized Analysis

## §5.1 What Is Amortization?

In the business world, amortization is when you spread out payments — if you have a big cost, instead of paying it all at once, you can spread it out over a long period of time. We'll do the same here. The issue is that for some algorithms (especially when we have data streams as input), we *usually* have small amounts of cost (that add up to $O(n)$), but once in a while we have a *huge* cost. So to measure the cost of such an algorithm, we average it out — so the expensive operation gets spread out and charged to all the cheap operations.

If the expensive operation could happen at any time — if the adversary could arrange the input so that it could happen operation after operation — then there's nothing we could do about it, and our algorithm is just slow. Amortization is useful when there's a relation between the cheap and expensive operations — if in order to set up an expensive operation, the adversary would have to first do a bunch of cheap ones, then we could spread out the cost of the expensive ones.




If the adversary could make us do $n$ all the time, this would be bad. But if they'd have to do $n$ 1's first, then we could spread out the $n$ to the 1's — we could replace them all with 2's instead.

## §5.2 Union Find

**Problem 5.1.** We want to maintain a pairwise disjoint collection of sets $\mathcal{S} = \{S_1, \ldots, S_r\}$, where each set $S_i$ has an arbitrary *representative* $\text{Rep}[S_i]$. We should be able to support three operations:

- MakeSet($x$), where we add $x$ to a new set and add that set to the collection.
- FindSet($x$), where we find the representative of the set that contains $x$.
- Union($x, y$), where we take two sets and rearrange them to make one set (and choose a representative).

Why would you want to do this? There are applications in the real world. One is maintaining the connected components of a graph as you perform a set of operations that may merge connected components to make a completely connected graph — we'll see that later when we look at minimum spanning trees.

Today we'll run through two different ways of implementing this. We'll proceed by starting with something simple that doesn't work very well. Then by analyzing it and seeing where it doesn't work well and why, we'll get ideas for how to improve it. This is a common theme in this class — we want to learn a process to go through to *find* the answer if it doesn't just pop into your head fully formed (which rarely happens). This is a good process — try something out to find what's hard about the problem, and then find a new algorithm or modify the one you have to resolve those problems.

### §5.2.1 A Doubly-Linked List

Suppose we store each set as a linked list, and Rep$[S_i]$ is the element at the *head* of the list. When we're handed an element $x$, we'll assume we're handed a pointer to it — so that we can actually find it in the list.

Let's suppose we have $\{3, 5, 9\}$ with representative 3 — as a doubly-linked list, this looks like

$$3 \leftrightarrow 5 \leftrightarrow 9.$$

Now let's suppose we have another set $\{6, 8\}$, stored as a list

$$8 \leftrightarrow 6 \leftrightarrow .$$

Now let's try to find the worst-case runtimes:

- For MakeSet$(x)$, we just have to plop down one element — that takes $\Theta(1)$.
- For FindSet$(x)$, we'd have to follow the pointer to $x$ — that's 1 step — and then follow the pointers forward to the head. In the worst case, all our elements are in one long list, and the adversary asks us for the element furthest from the head — that's $\Theta(n)$.
- Suppose we want to take union$(5, 6)$. Then we could start at 5 and 6, walk down to the tail of 5's list and the head of 6's list, and concatenate the list. This is kind of cute because we don't have to do any work to tell $\{6, 8\}$ that 8 isn't its representative anymore — the structure of linked lists takes care of that automatically. But the worst case runtime is again $\Theta(n)$ — we might have to walk all the way to the tail of a long list.

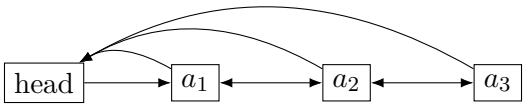
So MakeSet is cheap, but any FindSet or union could be expensive. So if you were the adversary and you wanted to make this algorithm look bad, you'd first make us make a really long list, and then ask us to keep finding the last element forever. So if $m$ is the total number of operations and $n$ the number of MakeSets, then if $m \geq n$ we could get $\Theta(mn)$ by first making a bunch of sets and stringing them together using union.

This is sort of like $n^2$, which is not good. But as hinted before, we want an expensive operation to require a bunch of cheap ones. Here what's bad is that one of the expensive operations is FindSet — you can do that a million times in a row without doing any work in between. If we can make FindSet cheaper, then we may be able to use the relationship between MakeSet and union to get a small *average*.

### §5.2.2 Making FindSet Cheaper

If we make FindSet cheaper, then we need to be able to get to the head more quickly.

So let's just augment our data structure — for every element, let's make a pointer to the head.




We'll also need to go to the tail in order to take unions, so let's add a pointer to the tail as well.

Now MakeSet$(x)$ still takes $O(1)$. And now FindSet$(x)$ is also $\Theta(1)$ — we can just jump to the head! That's a big improvement, because that was the most dangerous operation.

Now what happens to union$(x, y)$? We need to find the head of the set containing $x$ and the tail of the set containing $y$ — that's constant time — and concatenate them — that's also constant time.

But now when we concatenate, we have two problems. One is that we have an extra head and tail pointer — that's easy to fix. But the bigger problem is that now we need to change all the head pointers of everything in the second list. So this is still $\Theta(n)$ — the worst-case is when there's 1 element in the first set and $n - 1$ in the other, and we need to make $n - 1$ updates.

**Student Question.** *Why can't we just draw an arrow from the second head-pointer to the first?*

**Answer.** This may be possible to implement, but this takes a structure and morphs it into a structure that doesn't fit the pattern anymore. In a proof, that creates a bunch of cases — when there's one hop to the head, or two, or three or four. And this gets very complicated. We tend to try to avoid things like that, because it gets into the messy realm instead of the elegant realm. This may have reasonable behavior, but we'd have to think about it for a long time to figure it out.

How good is this? The adversary can do something kind of nasty — they can make us do MakeSet(0), ..., MakeSet($n-1$), and then merge them in an unpleasant way — union$(1,0)$, union$(2,0)$, ..., union$(n-1,0)$. Then we have to move a bunch of head-pointers every time, so we end up with total time

$$\Theta\left(n + \sum_{i=1}^{n-1} i\right) = \Theta(n^2).$$

So this is not really an improvement — it's harder for the adversary, but still not great.

The problem is that the adversary keeps asking us to tack one element onto a really long string, but then we have to update all the head pointers for the really long string. Of course, there's a better way to do this union — we should always stick the *shorter* set onto the end of the *longer* set, so that we have less than half of the head-pointers to update (instead of $n-1$ of them). The problem doesn't tell us how we need to choose the representative — it's just the way we wrote the algorithm that told us to tack one onto the other in the order they're given.

So instead, let's tack on the shorter set to the longer one.

Now the adversary wants equal-sized pieces, so that no matter what we do, we have to move half the pointers.

For example, let's suppose we'll have 8 elements. Then the adversary can have us make two elements and union them to make a pair, and then do the same with two more elements, and then make us union this to get a 4-mer. Then we do the same to find another 4-mer, and union those to make a 8-mer.

Now we see that *we can't do a union until we've done a certain number of makesets* — in order to do an expensive operation, we have to first do a bunch of cheaper ones. In this example, we have $n = 8$ and $m = 15$ — we have 8 elements, and we do 8 MakeSets and 7 unions.

**Proposition 5.2**

The total cost of all union operations is $O(n \log n)$, and therefore the total cost is $O(m + n \log n)$ — so the cost per operation is $O(\log n)$.

This $m + n \log n$ is better than the $n^2$ we had, and $\log n$ is better than $n$.

**Student Question.** *How do we know the length?*

**Answer.** We augment our data structure again — in MakeSet we introduce the length as 1, and in union we add the lengths.

*Proof.* We'll pick some element and watch it through the entire process, and then we can simply multiply by the number of elements. So let's focus on an individual element $u$.

It's created with MakeSet, at which point $|S(u)| = 1$. Then whenever $S(u)$ merges with another set $S(v)$, one of two things happens:

- If $|S(v)| \geq |S(u)|$, then $u$ is in a smaller set, so its head-pointer is updated; this bears a cost. But then the length of the set containing $u$ changes by at least doubling — $|S(u)|$ is at least twice what it was before.

- If $|S(v)| < |S(u)|$, then $u$'s head-pointer is not updated, so there's no cost from $u$. In this case, $|S(u)|$ still grows (by at least 1).

So every time we pay for $u$, the set at least doubles! But if there's only $n$ elements in the dataset, then we can double the size of this set at most $\log_2 n$ times. So the maximum cost of unions for $S(u)$ is at most $O(\log n)$, since $n$ is the number of elements in the data structure.

There are $n$ elements just like $u$, so the total cost of *all* unions is $O(n \log n)$, and the total cost of all operations is $O(m + n \log n)$. Then since $m \geq n$, the amortized cost per operation is

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

## §5.3 Amortized Analysis

**Definition 5.3.** An operation has amortized cost $T(n)$ if and only if any sequence of $k$ operations have cost at most $kT(n)$.

So this is still worst-case analysis — the adversary can produce the nastiest sequence, and our algorithm still has to take at most $kT(n)$ time.

One common method of doing this is the **aggregate method** — what we did here.

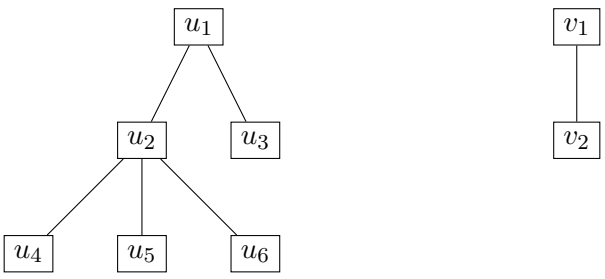
In the book and recitation notes, you'll also see the **accounting method**. Imagine our data structure; as you do an inexpensive operation, you imagine you're putting in a coin. But you overpay — if it costs 1 coin, you might put in 4 coins — 1 to add it to the data structure, and the other 3 are carried with the element. Then in the future, when that element undergoes more changes, you pay for those changes with these coins. It's sort of like paying up front.

The conceptually most difficult method is called the **potential method**. That's similar in some ways — when you do a cheap operation, you store some extra cost that you can expend later when you do an expensive operation (so you've saved up for it in some sense). But instead of associating coins next to *each* element, you store it in a potential function associated with the data structure. (This is a little like potential energy — if a ball goes down a ramp and then climbs another ramp, it slows way down, but it has a bunch of potential energy stored up that can be released on the next ramp.)

## §5.4 Improving UnionFind

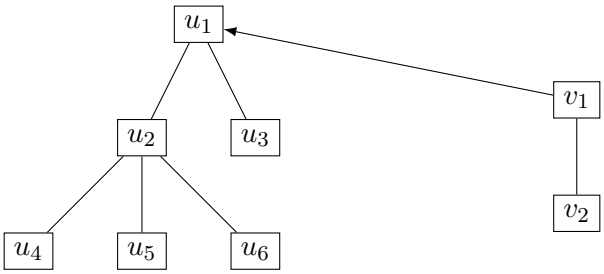
We've done as much as we can with the linked list; let's try something different.

We don't actually need the elements strung out in an array — we just need them to connect to the representative. So instead, let's think about a forest of trees — where a set is represented as a tree, and the representative is the root of the tree.




Then if we're asked to do MakeSet($u$), we simply initialize a tree with a single node and make it the root, which takes constant time.

When we're asked to do a FindSet($u$), we start where we're told to start, and climb pointers to the root of the tree. In the worst case, the time this takes is the height of the set containing $u$. To do union($u, v$), we can do two FindSets, and then connect the root of the $v$-tree just under the root of the $u$-tree — that takes time $\Theta(S(u) + S(v))$.




Now we'll have the same worst-case behavior as before — the adversary can force us into a situation where we create a really long, gangly tree, and we keep having to do unions that make us climb up from the bottom of the tree. Then everything's the same as the linked list from before — so this starts out badly.

But as we've learned before, we don't need to let the adversary make us do unions in a crazy way, where we have to merge the taller tree into the shorter tree.

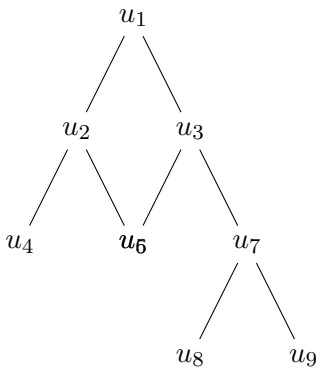
So the first idea is to merge the shorter tree into the taller tree. Then instead of having long, stringy trees, we have branched, bushier trees, that are shorter overall.

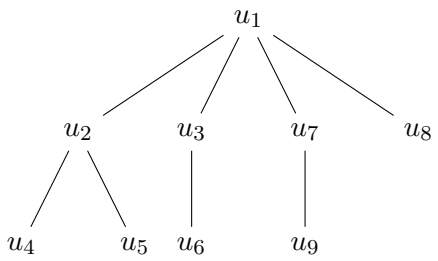
Then this becomes $O(\log n)$ cost for the union, on average. So that works out pretty well.

But can we do better? The adversary makes us climb through the whole tree, and we might have to do this multiple times. But instead, every time we have to climb a tree and find the representative, then we can climb again, and *directly connect everything we find to the beginning*!

Now the adversary probably doesn't want us to do the same thing again — does this limit the adversary's ability to cause us trouble?

Suppose we have



Suppose we do FindSet($u_8$). Now when we connect $u_7$ to the root, all its child nodes get carried along with it — even though we didn't traverse $u_9$, it came closer to the root anyways.

**Proposition 5.4**

The amortized cost of $m$ operations, $n$ of which are MakeSet, is $O(\log n)$.

*Proof.* We'll prove this with a potential function $\Phi$ from a configuration of the data structure to $\mathbb{R}$, and we'll let $\hat{c}_i$ — the amortized cost of operation $i$ — to be

$$\hat{c}_i = c_i + \Delta\phi,$$

where $\Delta\phi$ is the difference of the potential before and after $i$. Then

$$\sum_{i=1}^{m} \hat{c}_i = \sum_{i=1}^{m} c_i + \phi(m) - \phi(1),$$

since the potential differences form a telescoping series. There's a rule that the potential is the extra stuff we've stored up; so we should require that at no point is the final potential less than the initial.

Here let's define

$$\phi(\text{DS}) = \sum_u \log_2(\text{size of the subtree rooted at } u).$$

We want the potential to be big and positive, when the data structure is poised that it could have to execute an expensive operation (we want a lot of savings before something that's expensive enters your life).

For MakeSet, we have $\hat{c} = 1 + (\log_2 0 - 0) = 1$.

For Union, we need to perform two FindSets, and then link. We are out of time, so the remainder of the proof is in teh recitation notes. □

This also comes out to $O(\log n)$, but if we combine the two ideas — merging the shorter tree into the longer tree, and flattening the tree — then we get something that's almost constant.

# §6 Competitive Analysis

We'll see that competitive analysis is used for a specific type of situation. Usually, we care about the absolute worst-case runtime of an algorithm. In a competitive analysis, the actual runtime isn't our goal; instead, we want to compare to the runtime of some other algorithm. Why? If the algorithm we're comparing to is the theoretically best possible algorithm you could ever have, and we're only a multiplicative constant away from that, then we have the same order-of-magnitude runtime as the fastest possible runtime. That's actually better than our usual analysis! When we usually find an algorithm that's $O(n^2)$, we don't know if we've found the best runtime — what if there's an $O(n \log n)$ algorithm? It's useful to know whether we have the best algorithm — to know whether we should keep trying to improve our algorithm or not.

## §6.1 The Problem — Self-Organizing Lists

**Problem 6.1.** We are given a list $L$ that contains $n$ elements. There is only one operation, called Access($x$) — it takes as input the *key* of one of the elements in the list, and finds the element with that key. We start at the head of the list and walk through the list until we find an element with that key, so this costs us $\text{rank}_L(x)$. After every Access, we can reorder the list via a mechanism of *transpositions* — we can swap adjacent elements, and we can do this multiple times for larger rearrangements. Our goal is to choose a sequence of transpositions that optimizes the performance of Access by minimizing the cost $C_A(S)$ over all sequences $S$, in an online manner.

By "over all sequences," we mean the *worst-case* sequence.

**Example 6.2**

Suppose $L$ is the following:

| 12 | 3 | 50 | 14 | $\cdots$ | $x^{(L)}$ | $\cdots$ | $x^{(n)}$ |
|---|---|---|---|---|---|---|---|

Then Access(14) would cost us 4. If we then decide to transpose elements 2 and 3 — the 3 and the 50 — that costs us 1, so the total cost is 5.

**Definition 6.3.** An **on-line algorithm** is one where we only see the input sequence one element at a time — we have to act on the first element before we see the second, and so on.

In contrast, an **off-line algorithm** is one where you're given the entire input stream, and can optimize your sequence of transpositions *looking* at what elements you'll have to access in the future.

When we think about worst-case analysis, we can think of an adversary. It would be pretty bad if the adversary tried to access the last element every time.

We need to specify what the adversary is allowed to know. But we don't actually *need* an adversary — we care about the worst-possible behavior is, and the adversary is just a theoretical construct. So the adversary should know our algorithm — since by luck, someone could keep picking the last element. So our adversary has to know everything (unless we have a randomized algorithm — and even then, the random number generator could output the worst-case scenario, and that's why we do average case analysis instead).

If the adversary keeps picking the last element, that costs at least $|S| \cdot n$. We can't make this better by doing transposes — that just costs us more. So then the worst-case cost is $\Omega(|S| \cdot n)$. This would be quite boring, but it can happen.

What if we watched the input stream for a long time, and we knew how many times it accessed each of the keys? If we knew the probability distribution, we might be able to find a smart ordering, and then just leave it there — suppose $p(x)$ is the probability that Access$(x)$ occurs. Then if we don't do any rearranging,

$$\mathbb{E}[C_A(S)] = |S| \cdot \sum_{x \in L} p(x) \cdot \text{rank}_L(x).$$

We can minimize this by arranging the list so that the $p(x)$'s decline as we get further from the head.

That's an intuition for the problem — we'd like the more accessed keys near the front, and the less accessed keys near the back. We could try to do this even with an online algorithm — keeping track of the frequencies for some initial period, and then shuffling the list according to those frequencies.

## §6.2 A Practical Algorithm

Empirically, it's been found that the move-to-front algorithm (mtf) yields good results.

**Algorithm 6.4 —** After accessing $x$, we move $x$ to the head of $L$.

Then the cost of Access$(x)$ is $\text{rank}_L(x)$ for the access (where this is the current position), and $\text{rank}_L(x) - 1$ for the transpositions (since we have to move forward $i - 1$ positions). So this means the total cost is $2\,\text{rank}_L(x) - 1$.

We'll find that this is actually within a multiplicative constant of the *best* you can do, *even if you cheat*!

## §6.3 Competitive Analysis

Competitive analysis was developed by Sleator and Tarjan in 1985.

**Definition 6.5.** An **on-line algorithm** is $\alpha$-competitive if there exists a constant $k$ such that for any sequence $S$ of operations, $C_A(S) \leq \alpha \cdot C_{\text{OPT}}(S) + k$, where OPT is the optimal *off-line* algorithm.

So we're handicapped with an on-line mode and OPT gets to see all the input, and we still have to be within a multiplicative constant of OPT.

**Theorem 6.6**

MTF is 4-competitive for self-organizing lists.

The main takeaway for the rest of the lecture is how we can perform analysis in this competitive framework — the tools and thought processes we have.

*Proof.* Let $L_i$ be MTF's list after the $i$th ACCESS — so $i$ is the step we just finished. Let $L_i^*$ be OPT's list after the $i$th access. Let $C_i$ be MTF's cost for the $i$th operation, and $C_i^*$ the same quantity for OPT. We've already calculated $C_i$ — it's

$$C_i = 2\,\text{rank}_{L_{i-1}}(x_i) - 1,$$

since the list before this step was $L_{i-1}$. Meanwhile, we know

$$C_i^* = \text{rank}_{L_{i-1}^*}(x_i) + t_i,$$

where $t_i$ is the number of transpositions OPT chooses to do after the $i$th access.

Now we'll use an amortized analysis with a potential function. We're trying to not compare the runtime of MTF, but to compare it to OPT. So our potential should involve a comparison of what the two algorithms are doing, and what their lists look like — if the current state of the list is $L_i$, define

$$\phi(L_i) = 2 \cdot \#\text{inversions between } L_i \text{ and } L_i^*,$$

where the number of inversions is the number of pairs

$$\{(x, y) \mid x <_{L_i} y \text{ and } y <_{L_i^*} x\}.$$

By this funny notation, we mean $\text{rank}_{L_i}(x) < \text{rank}_{L_i}(y)$ and $\text{rank}_{L_i^*}(y) < \text{rank}_{L_i^*}(x)$.

**Example 6.7**

Suppose that at some point, $L_i$ is the list *ecadb*, and $L_i^*$ is the list *cabde*. Then the pairs in $L_i$ are $(e, c)$, $(e, a)$, $(e, d)$, $(e, b)$, $(c, a)$, $(c, d)$, $(c, b)$, $(a, d)$, $(a, b)$, $(d, b)$ (in the order in which they occur). Then we can run through $L_i^*$ and see whether the ordering is the same — in which case it's not an inversion — or flipped — in which case it *is* an inversion.

We can see that $(e, c)$, $(e, a)$, $(e, d)$, $(e, b)$ are all inversions — they're flipped in $L_i^*$. But $(c, a)$, $(c, d)$, $(c, b)$, $(a, d)$, $(a, b)$ are not inversions — they occur in the same order in $L_i^*$ Finally, $(d, b)$ is an inversion.

So we have exactly 5 inversions in this example.

The number of inversions is the number of transpositions we'd need to turn $L_i$ to $L_i^*$ — for instance, in this example we could swap $d$ and $b$, and then swap $e$ with everything to move it to the end. So it takes 5 transpositions to convert one list into the other.

In any proof, there's a few things we should do right away.

**Claim —** At all times in the algorithm, $\phi_i \geq \phi_0$.

We've seen that one convenient way of showing this is by having $\phi_0 = 0$, and $\phi_i$ always nonnegative.

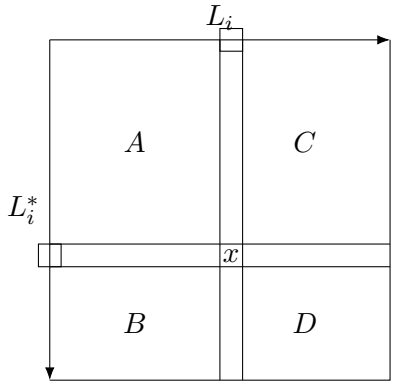
In this case, OPT and mtf start with lists in the same order, so $\phi_0 = 0$. Then we can't have a negative number of inversions, so $\phi_i \geq 0$ at all times.

The other property of a potential function we want is that it should be large and positive when we perform a step with large cost, so that we have wiggle room.

Because we're doing a comparative analysis, we care less about the cost of either algorithm, and more about the difference between them. If the lists are very, very different, then it's possible that OPT has a very easy one, and MTF a very hard one. Meanwhile, if the lists are identical, the costs have to be much more similar — the difference between them is due to transpositions only. So our potential is large when we have the *possibility* of a big cost, which is good.

Somehow, we have to relate costs and transpositions to the ordering of the list. So we need a picture or framework that relates this all together. Sleator and Tarjan thought about this for a long time; the picture that comes out fits the idea, but how you come up with it is a creative process.

Let's look at step $i$, and suppose we're trying to access some $x$. Now we draw a matrix:




Define $A$ as the set of elements before $x$ in both $L_{i-1}$ and $L^*_{i-1}$, $B$ the set of elements before $x$ in $L_{i-1}$ and after $x$ in $L^*_{i-1}$, and so on.

Then let's project up to $L_i$ — we have $A \cup B$ on the left, and $C \cup D$ on the right of $x$. Let $\text{rank}_{L_{i-1}}(x) = r$.

Then we can do the same for $L^*_{i-1}$ — here before $x$ we had $A \cup C$, and after $x$ we had $B \cup D$. Let $\text{rank}_{L^*_{i-1}}(x) = r^*$.

Then we know that on ACCESS, $r = |A| + |B| + 1$, and $r^* = |A| + |C| + 1$.

When we move $x$ to the front, $x$ is now before every element in $A$. So we've now created $|A|$ inversions. Meanwhile, we've *destroyed* $|B|$ inversions — we initially had inversions with all of $B$ ($x$ was after $B$ in $L_i$ and before in $L^*_i$).

Meanwhile, OPT might also move elements around. We don't know what it does, but know each transposition cannot create more than one inversion — so the biggest change that OPT can make is $t_i$. So then

$$\phi(L_i) - \phi(L_{i-1}) \leq 2(|A| - |B| + t_i).$$

Now the amortized cost of a single step in MTF is

$$\hat{c}_i = c_i + \phi(L_i) - \phi(L_{i-1}).$$

We've seen that $c_i = 2r - 1$, so

$$\hat{c}_i \leq 2r - 1 + 2(|A| - |B| + t_i).$$

Meanwhile, we also have an equation relating $r$, $|A|$, and $|B|$, so we can eliminate one of them — we have

$$\hat{c}_i \leq 2r - 1 + 2(|A| - (r - 1 - |A|) + t_i) = 4\,|A| + 2 + 2t_i.$$

So we made another variable disappear! This is important.

Now let's look at OPT's algorithm. To get rid of $A$, note that

$$r^* = |A| + |C| + 1 \geq |A| + 1,$$

so we can write

$$\hat{c}_i \leq 4(r^* + t_i).$$

This is exactly 4 times the cost of the $i$th step in OPT's algorithm! So $\hat{c}_i \leq 4c_i^*$ for each step — the amortized cost of each step in our algorithm is at most 4 times the actual cost in OPT's algorithm.

Now we're basically done. More explicitly, we have

$$\hat{c}_{\text{MTF}} = \sum_{i=1}^{k} \hat{c}_i = c_{\text{mtf}} + \phi_{\text{final}} - \phi_{\text{init}},$$

which means that

$$c_{\text{MTF}} = \hat{c}_{\text{MTF}} + \phi_{\text{init}} - \phi_{\text{final}}.$$

But since we know $\phi_{\text{init}} = 0$ and $\phi_{\text{final}} \geq 0$, this means

$$c_{\text{MTF}} \leq \sum_{i=1}^{k} \hat{c}_i \leq \sum_{i=1}^{k} c_i^* = 4c_{\text{OPT}}.$$

So we are done — for any sequence of operations $S$, the total cost of MTF is no more than 4 times the total cost of OPT (and our additive constant $k$from the definition is 0). □

## §6.4 A Few Thoughts

Today we learned a new kind of analysis. We used amortized analysis almost as an intermediary — to get to something that didn't have any amortization in it. And we learned a new framing, or a new way to think about algorithms, that adds to our arsenal.

We compared ourself to an omniscient algorithm OPT that we *never had to define* — we could prove things about how good we are relative to a theoretical construct that we never needed to define, and that's really powerful.

This is a fun problem to play with — you might go back through it and ask what if the cost of a transposition was 2 instead of 1 — would that change anything? What if a transposition was free? You might go back to the potential function and ask why we have a 2 there — what if we had 1, or 50?

# §7 September 27, 2022 — Hashing

## §7.1 Hashing Recap

### §7.1.1 The Dictionary Problem

A dictionary is a data type supporting the following operations:

- INSERT($key, item$) — which adds the item to the set with the given key. (We usually care about the key, not the item.)
- DELETE($key$), which deletes the item with the given key.
- Most importantly, SEARCH($key$), which finds the item with the given key if it exists.

Hashing is a very efficient implementation, under some assumptions — we need the key to be hashable.

We'll mostly focus on SEARCH — once we know how to search, insert and delete are straightforward.

**Definition 7.1.** A **hash function** $h$ maps the universe $U$ of all keys into $\{0, 1, \ldots, m-1\}$.

**Notation 7.2.** We use $U$ to denote the universe of all possible keys, and $K$ to denote the set of keys we want to store. We let $|K| = n$.

We know $K \subseteq U$, but generally $K$ is much smaller than $U$. The size of the hash table should be roughly comparable to the number of keys we want to store — if it were much larger then we'd be wasting space, while if it were much smaller then we'd have many collisions.

Collisions are the bane of hashing. If there were no collisions, we could simply store $k_1$ in $h(k_1)$, $k_2$ in $h(k_2)$, and so on. But if $h(k_2) = h(k_5)$, we can't be storing $k_2$ and $k_5$ in the same place.

So when we're doing hashing, we want:

- A hash function $h$ that mininmizes collisions;
- A way of dealing with collisions.

Today we'll focus on one of the simplest ways to deal with collisions — by *chaining*. If we're trying to enter multiple items into the same slot, then we instead insert them as a linked list $\bullet \to 49 \to 89 \to 52$.

There are other ways (for example, probing — if we have a collision, then we try some other entries in the table), which each have pros and cons. The disadvantage of chaining is that a linked list has many pointers, which wastes space; but it's the cleanest, so we'll focus on it for now.

We'll typically assume that the cost of evaluating $h(k)$ for a given key is $O(1)$ — we usually just evaluate some arithmetic expression. What typically takes more time is then scanning the linked list we've stored in our hash table $T[h(k)]$ to see whether $k$ is in the list. In general, this takes $O(\text{list length})$ — or equivalently, the number of collisions in our entry.

This is why we really care about the average number of collisions.

We know that under the **simple uniform hashing** assumption, the *expected* number of collisions is $\alpha = n/m$. So if the number of keys and size of the hash table are roughly the same, then our search time is constant in expectation. (The actual time depends on the list length, which is a random variable depending on the number of collisions.)

**Remark 7.3.** There is a tradeoff here — a binary search tree gives a deterministic algorithm, but an extra $\log n$ factor. Meanwhile hashing gives you a constant, but only in expectation.

### §7.1.2 The Simple Uniform Hashing Assumption

**Definition 7.4.** The **simple uniform hashing assumption** is that the hash value of each key is random and independent — each key $k \in K$ is equally likely to be hashed to any slot of table $T$, independently of where other keys are hashed.

A major problem is that this assumption is not very realistic — what would it mean for a hash function to satisfy this assumption? In principle, we can generate the hash function at random from some family, not knowing what the key is.

For example, suppose $U = \{1, 2, \ldots, u\}$, and we want to hash 1 — so we want to calculate $h(1)$, which is supposed to be a completely random number. We might also want to hash 2, so $h(2)$ should also be a completely random number. And so on — so to represent this completely random function, we'd have to be able to represent the set of random numbers $h(1)$, $h(2)$, $\ldots$, $h(U)$. But this set is as big as the universe itself — so you'd need to store $|U|$ random numbers.

Of course, this is not feasible, and no one does it. Instead, we use some deterministic hash function, chosen to typically minimize collisions, and then we use this assumption to *analyze* what's happening. But the hash functions we use in practice don't actually satisfy this assumption — there's a gap between the actual functions we use and reality.

**Student Question.** What does it mean by a deterministic hash function?

For example, one popular hash function is $h(x) = x \pmod m$. This is completely deterministic — there's no randomness.

**Student Question.** What would a random hash function be?

To have a random hash function, we'd need to have some class of hash functions, and select one at random at the beginning of the process. For example, we might have a family of hash functions $h_a(x) = ax \pmod m$, and $H = \{h_a \mid 0 \le a \le m-1\}$. Then we can reason that the search time is in expectation over all choices of the hash function.

## §7.2 Universal Hashing

The concept of universality applies to a *family* of hash functions.

**Definition 7.5.** Let $H$ be a collection of hash functions, each mapping $U$ to $\{0, \ldots, m-1\}$. We say $H$ is **universal** if for all $x, y \in U$ with $x \neq y$, we have

$$\mathbb{P}_{h \in H}\{h(x) = h(y)\} = \frac{1}{m}.$$

Note that this probability is defined over a random selection of a hash function $h$ from the family $H$. This is a property of the *family*, not the hash function — if we take a *random* hash function from this family, we have a low probability of collision (so the vast majority of functions don't make $x$ and $y$ collide, although some may).

Note that this is a much weaker assumption than SUHA — if SUHA holds, then $h(x)$ and $h(y)$ are independent, so the probability they collide is $1/m$. This is much more modest — this doesn't say that if we have *three* keys, then $h(x)$, $h(y)$, and $h(z)$ are independent.

That's the power of this definition. On one hand, we'll see that with such a hash function, we can still guarantee constant expected search time; and at the same time, it's possible to construct such hash functions, which don't suffer from the huge representation size problem. So these are good enough to be useful and modest enough to be constructible.

**Theorem 7.6**

Let $h$ be a hash function selected uniformly at random from a universal set $H$ of hash functions. Suppose $h$ is used to hash $n$ arbitrary keys into the $m$ slots of a table $T$. Then for a given key $x$, we have

$$\mathbb{E}[\#\text{collisions with } x] < n/m.$$

So we can get the same result as the one we had under SUHA, but now we have much weaker assumptions.

*Proof.* Let $C_x$ be the random variable denoting the number of collisions between $x$ and the $n-1$ keys in $T$; we want to show $\mathbb{E}[C_x] < n/m$. To do this, we define indicator random variables $c_{xy}$, which are 1 if $h(x) = h(y)$ and 0 otherwise. Then the number of collisions is $\sum_{y \in T \setminus \{x\}} c_{xy}$, and by universality we know $\mathbb{E}[c_{xy}] = 1/m$ (since the collision happens with probability $1/m$). So by linearity of expectation,

$$\mathbb{E}[C_x] = \mathbb{E}[\sum c_{xy}] = \sum \mathbb{E}[c_{xy}].$$

Note that this holds always — we don't need the $c_{xy}$ to be independent. So this is $\sum 1/m = (n-1)/m$. □

This proof is really simple, but in a way it's quite deep — it demonstrates the power of both universality, and linearity of expectation. Universality does not mean that the random variables $c_{xy}$ are independent (in general, they will not be), but they don't *have* to be, because linearity of expectation works even if things are not independent.

**Student Question.** Every time we want to hash a new value, do we select a new function?

No — you select one at random from the family, and keep using it.

**Remark 7.7.** If an adversary sees what our hash function is, they can come up with nasty other keys; that would be bad.

So we do not need SUHA — universality suffices to get a constant expected number of collisions. But of course all of this relies on the fact that we can actually construct such families in a reasonable way; we will now see one such construction.

### §7.2.1 Constructing a Universal Family

Let $m$ be prime (we can do this because primes are fairly dense, so we can choose one that approximates the correct size). Decompose our keys $k$ into $r+1$ digits, each with value in the set $\{0, 1, \ldots, m-1\}$ — in other words, this construction works for keys $k = \langle k_0, k_1, \ldots, k_r \rangle$, where $0 \le k_i < m$.

**Example 7.8**

Recall our example of credit card numbers. For example, a credit card number 3457 1095 7423 1249 could be a key. We could represent it by taking each of the four parts as one of the "digits" — so $m \approx 10^4$ (it can't be exactly $10^4$ because $10^4$ is not prime, but we can select a prime that's a bit bigger), and $r = 4$.

This is not quite realistic because usually we want to store more than $10^4$ credit cards; so you could take two "digits" of 8 numbers each instead.

In order to use this approach, we need to represent our keys in this way. This isn't really a big deal — any key at the end of the day is a sequence of bits, and we can just chop the bits into pieces.

Recall that Radix Sort in 6.006 used a similar trick — sorting itnegers represented as $r+1$ digits.

Now we pick $a = \langle a_0, a_1, \ldots, a_r \rangle$ where each $a_i$ is chosen randomly from $\{0, 1, \ldots, m-1\}$ — for each digit, we pick a random number. Now we define the hash function

$$h_a(k) = a \cdot k = \sum_{i=0}^{r} a_i k_i \pmod{m}.$$

Then we take $H$ to be the family of all these $h_a$.

**Remark 7.9.** This may look familiar to the multiplication testing from earlier, where we took a random product with a binary vector. Here this is the same, except with a random $m$-ary variable, and the logic will be similar although more complex to what we saw in that lecture.

**Student Question.** Is $a$ fixed?

Yes, when we design the hashing scheme the first thing you do is select a random $h$, and then $h$ doesn't change. (IT's complicated because you also have insert and delete; maybe if you insert way too many elements you have to change the hash function. But in principle you keep the smae $h$ forever.)

**Theorem 7.10**

The set $H = \{h_a\}$ is universal.

*Proof.* We want to show that for any two different $x = \langle x_0, \ldots, x_r \rangle$ and $y = \langle y_0, \ldots, y_r \rangle$, the probability of collision is $1/m$.

Since $x \neq y$, there must be some position where they differ; without loss of generality we can assume $x_0 \neq y_0$. (They may be distinct at other entries as well.)

We have $h_a(x) = h_a(y)$ if and only if

$$\sum_{i=0}^{r} a_i x_i \equiv \sum_{i=0}^{r} a_i y_i \pmod{m}.$$

We can rewrite this as

$$\sum_{i=0}^{r} a_i (x_i - y_i) \equiv 0 \pmod{m},$$

and we can separate it as

$$a_0(x_0 - y_0) \equiv -\sum_{i=1}^{r} a_i (x_i - y_i).$$

Since $x_0 \not\equiv y_0 \pmod{m}$, it has an inverse mod $m$ (since $m$ is prime, so the numbers mod $m$ are a field), which means that

$$a_0 \equiv -(x_0 - y_0)^{-1} \sum_{i=1}^{r} a_i (x_i - y_i).$$

So we have rewritten what it means for $x$ and $y$ to collide in this form — all our steps are equivalences.

Finally, we can say what the probability of this happening is. To do this, we can imagine selecting $a_1$, $a_2$, $\ldots$, $a_r$, and then finally selecting $a_0$. Once we've selected $a_1$, $\ldots$, $a_r$, then no matter what choices we've made, we have a fixed number on the right-hand side; this means there's exactly one choice for $a_0$ that leads to the equality. So the probability that we choose this particular value of $a_0$ is $1/m$. □

Hashign works under certain assumptions; here we need the assumption that a decomposition $\langle x_0, \ldots, x_r \rangle$ can be easily done. If the keys are credit card numbers, this is natural — we have a long sequence of digits, and we split it into pieces where each piece has value roughly between 0 and $n$. In general, the range of our pieces will not be a prime, so we pick a prime slightly larger than what we want. We can alwyas think of keys as a sequence of bits — if the sequences are of a particular size, for example, 10110110111001110101, we can always just chop into appropriately sized pieces as $10110 \mid \cdots$, such that each piece ranges between 0 and $n$. In practice, when people use this, it's simplest when we have a fixed set of keys, so we know exactly what the number of keys is that we want to store; then we can pick a prime slightly larger than this. (How do you pick a prime? You can run over a bunch of values and check whether they're prime.)

> **Student Question.** Why do we need prime?
>
> In order for $x_0 - y_0$ to have an inverse, we need the numbers mod $m$ to be a field, which means $m$ must be prime.

> **Remark 7.11.** In reality, you won't have to code this — in Python the dictionary already does this.

So we've seen how to implement dictionary so that search works in expected constnat time under universal hashing, and we've seen how to impleent universal hashing in $O(r)$. So as long as $r$ is a constant (which is usually true), we get a dictionary that can be used to search in constant time.

## §7.3 Preview of Perfect Hashing

We have rigorous hashing with vonstnat expected time. But we'd like constant worst-case time. We cannot get that from the way we did hashing — collisions *can* exist. So certainly what we've done so far cannot guarantee a constant number of collisions.

But in another shocking developmentin the early 1980s, a more complicated concept called *perfect hashing* guarantees htere are no collisions whatsoever!

The basic idea is that we do hashing on two levels. The first level is exactly as before — we use a universal h ash function family to hash $n$ keys into a table of size $m$. Here we are going to have collisions.

What's new is after the first level of hashing, we do a second level — instead of storing all the elements that hash to an entry in a linked list, we create *another* hash table. This second-level hash table ensures that there are no collisions whatsoever.

Why is this possible? THe idea is that here, we only have to select a hash table for the three elements that land on this entry. To make sure that a perfect hash function exists, we can actually allocate an amount of space to be the square of the number of elements that you hash — for example, here we have 3 elements that hash here, so we allocate a 9-element table to assure that there is at least one hash function that has no collisions.

> **Student Question.** Before the inputs are given hwo do you know how much space to allocate?
>
> The best way of thinking of this is in the static case — wheere you're given the set of keys in advance, and for this set of keys you create the concoction os that you have no collisions at all.

In this setting, there is no insert or delete — suppose you are a networking company and you have a fixed set of IPs and you want an ultra-fast lookup of those IPs. You are given all these things in advance, and now you just want to be able to very quickly search.

So we select one hash function at random. Then we know exactly how many collisions there are. based on this, we know how much space to allocate for the second-level hash thing.

This construction crucially uses universal hash functions — you choose $h$ from a universal family in the top level, but more importantly on the second level you use a different hash function for every second-level table, and you have to be able to store this in the table. So each of these things needs a short description; that short description is guaranteed using the $h_a$ construction, since they're described using only $r+1$ numbers.

# §8 September 29, 2022 — More on Hashing

## §8.1 Recap of Universal Hashing

Last class, we discussed hashing, which solves the dictionary problem (mostly focusing on *search*). We will continue this approach until we discuss open addressing, when insertion and deletion is crucial — for perfect hashing there will be no insertion or deletion, and we focus on search.

Hashing is a super efficient way of implementing search. Last class, we saw the notion of *universal hashing*, which relies on a universal family of hash functions — a collection of hash functions $H = \{h_a\}$ such that if we select a hash function $h \in H$ at random, then

$$\mathbb{P}(h(x) = h(y)) = \frac{1}{m}$$

(where our hash functions map the universe to $\{0, 1, \ldots, m-1\}$). This is the best possible probability — if we hashed elements completely at random, then the probability they'd land in the same place is $1/m$, and this property is preserved by universal hashing (despite the fact that the hash function is not completely random — it's just random *enough*).

We then showed two things:

- If we have universal hashing — a hash function family with the universality property — then we can implement *search* (and *insert* and *delete*) to take constant expected time.
- We can construct a universal family of hash functions as long as the keys can be represented as $r$-tuples of integers in $\{0, \ldots, m-1\}$, that can be evaluated in $O(r)$ time.
- In conclusion, our dictionary data structure has constant *expected* search time, as long as $r = O(1)$.

But instead of having a constant *expected* search time, it's actually possible to guarantee constant *worst case* search time — that there are no collisions at all, using perfect hashing!

## §8.2 Perfect Hashing

> **Problem 8.1.** We're given a set of $n$ keys in advance. We want to construct a *static* hash table of size $m = O(n)$ such that SEARCH takes $O(1)$ time in the *worst case*.

Note that in this situation, we do not have insert or delete — we're given the keys in advance.

The main idea is the key picture for the construction: we have a universe $U$, and a set of keys $K \subseteq U$, with $n$ keys given in advance. In this construction, hasing proceeds at two levels. On the first level, we do something very similar to standard hashing — we select one hash function $h$ from the universal family, and take all our keys and hash them. The talbe has approximately the same size as the number of keys, so when we perform hashing, we are going to probably have collisions. In our example, one entry has 1, one has 2, and one has 3 things hashed to it. In the first level, we store the *squares* of these numbers.

Now we need some way of resolving the collisions. The key difference from hashing we saaw in the previous lecture is that instead of having a chained list that connects all hashed elements to our entry, we replace it by *another* hash table, with its own hash function. The table has size that's *quadratic* in teh number of

elements — if we're trying to store 3 element,s we have 9 entries. The hash function is selected *specifically* for this entry, and its only goal is to hash these three elemnets into a table of size 9, such that there are no collisions whatsoever. (We also store what hash function we're using in our first-level table.) We'll see that this is possible, and further more we'll see that even though we're obviously wasting space (ifw e had 100 elements, we'd have $100^2$ spaces in our table), the total size of everything is still linear in the number of keys.

So there's tow things we'll prove:

- perfect hasing on the secon dlevel — its possible to hash $t$ elements into a table of size $t^2$ with no collisions.
- In aggregate, we're not wasting too much space — the size of this hash table is still linear in $n$, although possibly with a large constant factor.

**Student Question.** Do we have the number of collisions in advance, or do we calculate it when setting up?

Everything's static here — for example, we want a quick way of determining whether a credit card is real. Then at least for a day, the entire dataset is there. So we calculate all the number of collisions beforehand, before doing the second-level hashing; this will be clarified in a moment. In effect, for starters we don't worry at all how quickly we can *construct* this thing; we do lots and lots of stuff, and at the end we output a hash table that works.

So for starters we will not worry about initialization time (although it will not be horrible) — we can think of this as doing it overnight.

**Student Question.** Is space $O(n)$ worst case or average case?

We'll prove expected; but that means the space is $O(n)$ with probability at least $1/2$, and that means we can keep trying. If we do all of this and at the end we used too much space, we can scrap everything and try again. Technically the algorithm that constructs this will be randomized, but the fact that there's an algorithm that succeeds with constant probability means that there exists a data structure like t his, and if we keep trying a large enough number of times, we will get one.

### §8.2.1 The Second Level

First we'll analyze what's happening at the second level. Suppose we have 3 items; then we hash them into a table of size 9 using a hash function selected from the universal family. We want to guarantee that it's possible to hash them without any collisions.

**Theorem 8.2**

Let $H$ be a class of universla hash functions for a table of size $m = n^2$. Then if we select a random $h \in H$ to hash $n$ keys into the table, the probability of one or more ocllisions is at most $1/2$.

*Proof.* By the definition of universality, the probability that any two given keys collide is $1/m = 1/n^2$.

Then since there are $\binom{n}{2}$ pairs of keys that could potentially collide, by the union bound the probability that *any* collision occurs is at most

$$\binom{n}{2} \cdot \frac{1}{n^2} < \frac{1}{2}.$$

□

In terms of arithmetic, this is a very simple argument; but it's very neat, and demonstrates the power of universality. Universality *only* covers the probability of collision for a *pair*; but that's sufficient because we can just take the $\binom{n}{2}$ pairs and union bound.

So now we perform a randomized algorithm:

- We select a random hash function $h \in H$ that maps our $n$ keys $K$ into a table of size $m = n^2$.
- We check if $h$ induces any collisions in $K$.
- We repeat until we find a $h$ with no collisions.

We succeed with constant probability at each step, so at the end of this process — when we exit the loop — we have found a hash function that has no collisiosn.

Then we take this hash function, and we plug it into our table, so that we know for future reference that it's the hash function we used to hash the elemnets in the smaller thing.

At this point, we won't worry about the runtime of the algorithm. But it's actually pretty good — essentially we've proved that whether each loop succeeds or not is a coin flip, adn the expected number of coin flips you need to get heads is a constnat. So the number of times you have to repeat this process is constant in expectation.

Observe that the fact that if we sleect a random hash function, we find one without collisiosn without constant probability, automaticlaly implies that such a hash function *exists* — if 50% of hash functions give you no collisions, then there certainly exists one, and we find one by "throwing random darts."

> **Student Question.** How long does the check take (when we select a random hash function)?
>
> Prof. Indyk was planning not to talk about runtime for the tiem being. But it relal ytakes time linearin the number of keys (with an asterisk), since lal we need to do is select $H$ at random, go key by key and see if there are any collisions (marking the stuff in the hash table). The asterisk is that this assumes our array that we allocated in the beginning is already nicely initialized. That is not a big deal, but in the worst case it takes quadratic time to initialize the array.

> **Student Question.** If we modified the problem to allow ourselves a size $n^2$ hash table, would we be done?
>
> Yes — this alone gives a solution to perfect hasing with size $n^2$. The reason to have two levels is to reduce the total space to linear; but this theorem alone gives us a hash table of size $n^2$ with no collisions. In fact in recitations we'll see a variant which still has constant time but reduces the space to $n^{1.5}$. So if you can spare some space, this is already a pretty useful thing; the reaosn we use two levels is to reduce space (so that we only have quadratic blow up on the second level).

The expected runtime is $O(n)$ (plus the array size for initialization).

### §8.2.2 Analysis of Storage

What happens if we take this quadratic blowup and apply it individually, but only to the keys which collide in one of the entries of the top-level hash table? For the level 1 hash table, take $m = n$; and let $n_i$ be the rnadom variable for the number of keys that hash to slot $i$. Then the amount of space we need is $n_i^2$. So the expected total storage required is

$$\mathbb{E}\left[\sum_{i=0}^{m-1} \Theta(n_i^2)\right],$$

where this expectation is over the random choice of the top-level hash function $h$ mapping the keys into the first-level hash table.

At this time, what remains to be done is to bound

$$\mathbb{E}\left[\sum_i n_i^2\right].$$

The key observation is: let $C$ be the number of *pairs* of elements that collide in the top-level hash table. Then if we have $n_i$ collisions, the number of pairs is roughly $n_i^2$ (it's actulaly $\binom{n_i}{2}$). So we can swithc between $n_i^2$ and the number of pairs that collide at the $i$th slot —

$$\sum_i n_i^2 = O(C).$$

(We won't worry about places with 1 entry — there are at most $n$ things, so we can just add $n$.) This means it suffices to estimate $\mathbb{E}[C]$, which is much easier.

To do this, we use an approach almost identical to what we've seen in the previous lecture, just applied to all pairs. Write $C = \sum_{x,y} c_{xy}$, where $c_{xy}$ is the indicator variable for whether $x$ and $y$ collide — so $c_{xy}$ is 1 if $h(x) = h(y)$ and 0 otherwise. Summing all these indicators gives us the totla number of collisison. At this point, we are essentially done — note that $\mathbb{E}[c_{xy}] = 1/m$, so then

$$\mathbb{E}[C] = \mathbb{E}[\sum c_{xy}] = \sum \mathbb{E}[c_{xy}] = \binom{n}{2} \cdot \frac{1}{m} = O(n^2/m) = O(n).$$

(Since $\mathbb{E}[c_{xy}] = 1/m$ by universality, and we set $m \approx n$.)

So what we just showed is that even though we're wasting a bunch of space — if we had 6 elements we'd be allocating 36 spaces — in the expected sense, when we take expectation over the top-level hashing, it cannot be the case that all elements fall into one place for example (where we'd have to be allocating $n^2$ space). Essentially, the argument seen in this slide shows that in expectation, this cannot happen — the expected amount of space we allocate at the second level in total is still linear.

That pretty much concludes the proofs of the perfect hashing scheme.

> **Student Question.** What is the tradeoff with using universal hashing?
>
> The benefits of perfect hashing are clear — worst-case constant time. But the tradeoff is for starters, this only works in teh *static* case — the user has to be nice enough to give you all the keys and then make no changes. That's one disadvatnage. The second disadvantage, not really captured by our analysis, is that the total space is linear, but the constant factors are worse. We can see why — in the standard hash table we don't have to store the number of elements, or the parameters of hte hash functions, nad we're not wasting a bunchof space. This adds overhead — we've proven that we're not wasting *too much* space, but we're wasting some. SO this would not be as space-efficient as if we just used a standard hash table.
>
> This approach (and many related developments) ar eparticularly useful in real-time situations — for example, if you have a router and in microseconds, you have to decide where to push. Then you can't rely on expectation, because you only have a fixed number of microseconds to make a decision. So when speed is absolutely crucial, you'd use this type of hash tables. In practice, if you don't really care about the difference between worst-case and expectation, you'd use the standard hash tables — you select hash functions at ranodm and live with teh collisions, using randomenss to mean that too many collisions don't happen too often. But then you are potenitlaly vulenrable to adversaries — if the user is adversarial, they may observe the behaviour of your hash table, see which items make the hash table respond slower ,and then by probing it's possible to figurte out which queries are bad and only issue those. This system does not have that vulnerability because once it is constructed, thereis no randomness.

**Student Question.** How does the adversary make sense in this situation where keys are static?

By choosing bad searches. The adversary in the normal case can figure out hwihc keys cause slow responses, and then keep asking those. But you could do more damage if you could insert and dleete keys (since then you could create a very long line); that is not possible here by definition.

**Student Question.** Why is $C$ an estimate on the size of the table?

This is because if we have $n_i$ keys in our table, then it has size $n_i^2$. But $n_i^2 \asymp \binom{n_i}{2}$, so up to constant factors, it's the same as the number of *pairs* of keys in this entry. So we're essentially just rewriting $\sum n_i^2$ using the more convenient $\sum \binom{n_i}{2}$.

### §8.2.3 Summary

We've constructed a static data structure, where $K$ is known in advance (this doesn't work with insertions and deletions but we can do this if we have keys in advance). The approach is two-level: first we throw in all the keys, but instead of chainign we replace the chain with an array of quadratic size. The first-level table size is $O(n)$ by design. The sum of sizes in teh second-level hash tables is also $O(n)$ (as we proved). So the space is $O(n)$, and more importantly, search only takes $O(1)$ time in the worst case! (Assuming that our universal hash functions can be evaluated in constant time.)

**Student Question.** Why is this worst case and not expected?

Technically our algorithm is randomized; it constructs a table with expected constant size. But if the size is too big and we don't like it, then we can simply drop everything and try again. This means we can make it a Las Vegas algorithm — keep trying until we succeed.

The search is necessarily $O(1)$. The space is $O(n)$ expected. But if it's too big we can drop everything and try again; so we can make this worst-case as well.

In a nutshell, we have a randomized algorithm that gets a perfect hashing table that guarantees constant time, and using the Las Vegas version, gives $O(n)$ space.

**Student Question.** Is it possible to do dynamic perfect hashig?

Yes but then the guarantees become even more subtle. You can't get all of insert, search, and delete in worst-case constant time. In particular binary sesarch trees are still useful — they are not constant, but everything is in worst case.

## §8.3 Open Addressing

Now we will change gears almost completely. So far, we've talked about how to optimize *time*. Here we will talk about optimizing *space*.

What does optimizing space mean? Universal hashing with chaining gives $O(n)$ space. But the main difference between what we discussed last lecture and open addressing is best seen in practice. It's a way to reduce space in practice — the difference is not asymptotic, it's up to constant factors. But still space is very useful, so most implementations of hashing use open addressing.

### §8.3.1 The Trouble with Chaining

Suppose we resolve collisions by chaining. This is clean and easy to analyze. But the issue is that we have a bunch of pointers, which takes memory. This won't show up in the asymptotic expressions, but it does

create problems in practice — you don't just store your items, but also the associateed links. This means you'r eusing *two* memory cells per item, instead of just one. (This is on top of the initial space.) So we are using extra links, which in principle doubles the amount of space we are using.

There are also more subtle issues, dealing with security. Generally speaking links can be problems with security because we don't know where they're pointing — if an adversary changes the links then there can be trouble. But we won't focus on this.

### §8.3.2 The Main Idea

The most popular way of fixing this is *open addressing*. (Prof. Indyk doesn't know where the name comes from.)

Instead of chaining, when a cell is occupied, we simply try again — we find another bucket.

What this means is if we have a hash table, say we are trying to hash 1 into the table. Then we try to hash 2 into the table, and for all we know, if we hash it, it might collide. In chaining, we'd just put 1 and 2 in a single linked list. In open addressing, we simply try inserting 2 in somewhere else. Maybe there's somethign else stored in side there, such as 3; then we try again somewhere else. We keep trying until we hit an empty spot. And that's essentially what open addressing is.

Formally, we no longer have a single hash function; instead it defines a *sequence* of probes. We have one value for the first probe, and if we succed we're done; otherwise we try the value for the second probe, adn so on. So we have a sequence $h(k,1)$, $h(k,2)$, ... that should cover all the buckets. We need $m \geq n$ for this to work (we can't store $n+1$ elements in an array of size $n$); then at some point we get an empty spot and we're done. Formally, $h$ is a function $h: U \times \{1,\dots,m\} \to \{0,\dots,m-1\}$.

So there are no lists, and collisions are resolved directly by moving somewhere else — this is how to do insert.

### §8.3.3 Search

Clearly, we have to use the probing sequence as well in the search — suppose we were searching for 2 (which got stored somewhere, say in its third spot we tried). Then we first use the first element of the probe, and see there's a 1 there. That's inconclusive, since 2 might be somewehre else. Then we use the second element, again we don't see 2 but that's inconclusive. Then we search the third element ahd hit 2, so we've found it.

Of course, we could in principle be searching for 4, which is not in the array .We might have such bad lluck that we hash 4 and don't find it, and then hash again and don't ifnd it, and so on .Then at some point we're going to hit empty space; once that happens, that means it's really not in our array, and we can declare that it's not there.

That is how we resolve insert and search in open addressing hashing.

> **Student Question.** Why can we stop at an empty cell?
>
> Notice that so far there is only insert and search. Once we get to deletion, things become more complictaed (as we will soon see). If we *only* have insert and search — no deletions — then this process makes sense.
>
> When an item is inserted, we keep trying until we hit empty space, and then we stop and actually put the elemnet there. SO if the item is in teh tbble, then when we search we're replicating the same process — when we search we keep hitting the same things that we hit when we did the insertion, (maybe even more because more stuff could've been inserted). Then by following the same probe, we must find this element if it's there.

**Student Question.** Does this only return whether the key is there and not the value?

You can also attach the items to the keys; then we can find them as well. Usually the items just tag along.

**Student Question.** What's the runtime?

This depends on teh ratio between the number of inserted keys and the size of the table. This is essentially because let's say we have $n$ keys, and a table of size $2n$. Then in that case, the runtime is something like 2 — assumign our hash function sare very nice — because when you want to find the next spot, there's a probability $1/2$ that it's empty. But if you have $n$ keys and allocate only $n \cdot 1.1$ space, you will have a huge amount of collisiosn and your time will shoot up.

### §8.3.4 Deletion

Deletions are somewhat messy precisely because of this conversation we just had. The most natural thing would be to probe until we find the item, and then remove it, leaving an empty bucket.

This doesn't work — for example, suppose we first insert 1, and then we try hashing 2 an it collides with 1 so we move it to a different place. And then suppose we delete 1. Now when we're searching for 2, we'll first see the empty space at 1, and we will conclude wrongly that 2 is not in the table.

So this is the mess that we have to deal with when we try to save space using open addressing.

So we need to use "tombstones." Whenever we delete an element, we leave it in the bucket, but with an extra bit that says that it was "deleted" — so if we inserted 1, and then we delete 1, instead of making it empty we just mark an X there — to say there was something there but now it's not there any more. The future search for $k$ sees that $k$ was deleted and inserts "not found". Then when we're trying to insert $k'$, if we hit the $k$ bucket, then we *overwrite* it with $k'$. So keeping the deleted items doesn't waste space, since we overwrite the deleted things when we want their space.

### §8.3.5 Efficiency

Now we want to see how efficient this is. It's clear that this is mroe space-efficient — we're not using any more pointers. (We'll ignore the tombstone bit since it doesn't cost that much.) Then to store $n$ items, we just use a hash table of size $m$, and it's up to us what this should be. Of course we need $m \geq n$, but other than that it's negotiable — the space is whatever we want it to be.

But the trdeoff is that if $n/m$ is too close to 1, then we're going to suffer in terms of runtime — under the *uniform hashing assumption* (that for each key $k$, the whole probe sequence $h(k,1)$, $h(k,2)$, ...is a random permutation independent of other keys), the expected number of probes is $1/(1-\alpha)$ for $\alpha = n/m$. So this gives us a tradeoff of optimizing space or time — to optimize time we need $\alpha$ small, and then we blow up space by a factor of 2. (But we are not storing pointers still.) If you mkae $\alpha$ too small though ($m = 10n$),then it may be more efficient to use chaining — you pay a it more for links, but you're not wasting 90% of your space.

In practice, there are hacks people use to form probe sequences (which do not satisfy the UHA, but are good enough).

# §9 October 4, 2022 — Greedy Algorithms: Minimum Spanning Trees

## §9.1 Greedy Algorithms

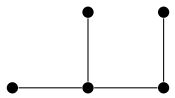
As discussed earlier and in 6.006, the key attribute of a *greedy algorithm* is that the overall problem becomes a series of subproblems, and we can look at each subproblem, ignore the overall problem, and just do the thing that seems best for this subproble; and somehow, this magically works in the overall problem as well. These are often optimization problems — you're searching for the minimum or maximum cost, and by doing something that's *locally* best (for example, to get the most weight you take the heaviest component and don't worry if that'll dead-end you), it magically works. In 6.006 we learned that there's special properties of the subproblem that make this work. You can prove those properties, but you can also just prove that the algorithm is correct without reference to those details.

For each subproblem, we make the "best" choice without consideration of the overall problem. There's a relationship between the subproblems and overall problem that's responsible for making the greedy choice the right choice. There'll be other problems where the greedy algorithm *doesn't* work — where the locally best choice won't be the globally best.

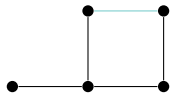
## §9.2 Some Definitions

**Definition 9.1.** A **tree** is a connected graph with no cycles.

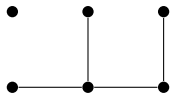
For example, the following is a tree:



Adding the blue edge would make this not a tree:



Similarly, adding an isolated vertex would make it not a tree:



**Definition 9.2.** A **spanning tree** is a subset of the graph's edges that form a tree containing all vertices.

In this class, we often learn about something and learn how to write an algorithm for it, and there's a step in between — to figure out some mathematical properties about the thing so that we can understand it better, and that might help us write an algorithm.

**Proposition 9.3**

If a graph has $n$ vertices, a spanning tree will have $n-1$ edges.

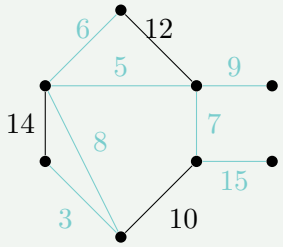
There could be lots of different spanning trees, but they'll all have in common that they have $n-1$ edges. If we had any more edges then we'd have a cycle, and if we had any fewer edges we wouldn't be able to reach all vertices, and it wouldn't be a spanning tree.

**Problem 9.4** (Minimum Spanning Tree Problem)**.** We are given a graph $G = (V, E)$, and edge weights $w: E \to \mathbb{R}$. We want to find a spanning tree $T \subseteq E$ with minimal total weight (where the weight of a tree is $w(T) = \sum_{e \in T} w(e)$.)

This is surprisingly useful in a lot of infrastructure development projects — for example, if we want to link up cities with cables, train lines, or airline routes; if we want to hook up houses with water pipes; and so on, and we want to minimize the cost of connecting everyone in some network. It also has other uses — some clustering algorithms (where we cut along the longer edges in the minimum spanning tree to create clusters), information theory (where we want to look for higher order relationships in information, and the minimum spanning tree gives a useful way of creating approximations for mutual information).

**Example 9.5**

Take the graph:




We can see some heuristics:

- We generally want to aviod large weights. But this isn't an absolute rule — 14 is avoided, but 15 is included.
- We generally like to include small weights — 3, 5 are the smallest weights, and they're all included.
- Some edges are unavoidable — for example, our edges of weight 9 and 15 are the only ways to get from those vertices to the rest of the graph.

In this case, the minimum spanning tree was unique. Meanwhile, we could take a simple graph, the square where all edges have weight 1 — this has four minimum spanning trees.

## §9.3 The Cut Property

**Theorem 9.6**

Suppose $G = (V, E)$ is a connected graph with cost function $w: E \to \mathbb{R}$. If we split $V$ into two nonempty groups $U$ and $V \setminus U$, then if $(u, v)$ is an edge of lowest cost (a "lightest edge") with $u \in U$ and $v \in V \setminus U$, then there is a MST containing $(u, v)$.

First, let's set some definitions:

**Definition 9.7.** A partition of $V$ into two nonempty, non-overlapping sets is called a **cut**.

**Definition 9.8.** An edge **crosses** a cut if one vertex is on one set of the cut, and the other is on the other side.

**Definition 9.9.** A cut **respects** a set of edges if no edge in the set crosses the cut.

Note that if $(u, v)$ is the *unique* lightest edge, then not only is it in *a* minimum spanning tree, but it is in *all* minimum spanning trees.

Now let's prove this. Today, many proofs will be about "cut and paste" — we'll look at two trees, cut a bit out of one and stick it into the other, and draw some conclusions. (This is also called an *exchange* argument).



*Proof.* Let's assume the theorem is false, so there is no minimum spanning tree that includes our light edge $(u, v)$. Let $T$ be a minimal spanning tree; then $T$ does not contain $(u, v)$.

Now let's suppose we add $(u, v)$ to $T$. This will create a cycle — there was already a path from $u$ to $v$ in $T$ (because a tree is connected), so if we add another edge, this creates a cycle.

But then there must be at least one other edge in that cycle that crosses the cut. Call that edge $(u', v')$.

If we now delete $(u', v')$, we again create a spanning tree $T'$ (since we've broken the cycle). But by assumption, $w(T') > w(T)$, since $T'$ is not a minimum spanning tree. On the other hand,

$$w(T') = w(T) + w(u, v) - w(u', v').$$

We were told that $(u, v)$ is a light edge crossing the cut, so we must have $w(u, v) \leq w(u', v')$. This means $w(T') \leq w(T)$, contradiction.

So then there is a MST containing $(u, v)$. □

## §9.4 Kruskal's Algorithm

**Algorithm 9.10 —** We initialize $T = (V, \emptyset)$. Then we examine the edges in increasing weight order — starting with the lightest and working our way up (breaking ties arbitrarily).

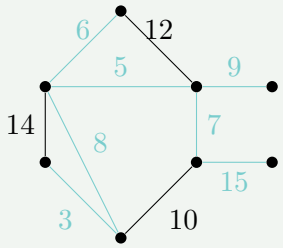
- If an edge connects two unconnected components, then we add the edge to $T$.
- Otherwise (if the edge would go between two parts of the same connected component), then we skip the edge (which would form a cycle).

We terminate when all vertices are in a single connected tree (or we could go to the end if we wanted to).

This explains the heuristic from our example — the reason we didn't take 10 or 12 or 14 is that it would form a cycle with edges lighter than it.

**Example 9.11**

Consider our example graph:




Now let's prove that this algorithm works. First, here's a proof that *isn't* correct.

*Flawed proof.* We'll consider a loop invariant — prior to each iteration, $T$ should be a *subset* of a minimum spanning tree (all its edges are part of the same minimum spanning tree).

At initialization, $T$ has no edges, so this is trivially true.

At all steps after that, edges added to $T$ are light edges crossing a cut — one way of thinking about this is that they connect two disconnected components, so we can take a cut that has one connected component on one side, and the other on the other side. So by the cut property, these edges belong to a MST.

At termination, all edges have been considered, and have been added to $T$ if they belong in the MST. So the final product must be a MST. □

What's wrong with this? The cut property says that the edge with this property (being a light edge crossing the cut) is in *some* MST. So you've collected a bunch of edges belonging to *some* MST, and put them into a big pile. But that doesn't necessarily mean they belong to the *same* MST!

**Remark 9.12.** One thing we should get out of this class is knowing how to construct proofs *correctly* — so that when we *think* we have a constant-time algorithm, we can be sure. In this case, the theorem is right, but our proof is wrong — so really, we have no idea whether the theorem is right or wrong. The message we should take from this is that for our lives, we should learn the skill of really convincing ourself that a proof is right. Even if we get all the points on a problem, we should read the solutions.

If the weights are *distinct*, then the MST is unique, and the proof works perfectly fine. But for non-distinct weights, our proof ends up in trouble. We'll see that the idea is to create a stronger theorem — that when we add an edge, we can be guaranteed that the new edge along with the existing edges is still part of a MST.

First, let's see how to implement this.

We have to keep track of these connected components as we go along — we have to remember which vertices are isolated, or part of dumbells, or flying birds or open squares or stars. This should scream UnionFind — we need to be able to ask whether two vertices are part of the same connected component, and phrased that way, the UnionFind data structure is natural to use.

We start with an empty tree, take each vertex, and do MakeSet($v$). Then we sort $E$ by weight $w$. We look at the edges in non-decreasing weight order, and ask whether it connects two vertices of the same or different connected components. If we construct this right — so that a set represents a connected component — then it suffices to check whether FindSet($u$) = FindSet($v$). If so, we ignore the edge; otherwise we add the edge $(u, v)$ to $T$. Then we need to perform Union($u, v$), to represent that we've connected $u$ and $v$ and therefore merged their connected components.

To analyze runtime, if we use the forest-of-trees data structure with both union by rank and path compression — then we saw MakeSet has constant time, and the amortized time of FindSet and Union is the *really* slowly-growing inverse Ackerman function:

- It takes $O(V)$ to perform all the MakeSets (since those can be done in $O(1)$ time each).
- It takes $O(E \log E)$ time to sort the edges.
- It takes $O(E \cdot \alpha(V))$ time to perform the FindSets and Unions.

We know $|V| - 1 \leq |E| \leq |V|^2$, so then our runtime is $T = O(E \log V)$.

## §9.5 A Stronger Cut Property

Now let's prove the stronger theorem we wanted — that if we have some edges that are a subset of a MST, we can identify a new edge to add so that we still have a subset of a MST.

**Theorem 9.13**

Given a connected (undirected) graph $G = (V, E)$ with a real-valued weight function on edges, let $A \subseteq E$ be a subset of edges included in *some* MST of $G$. Imagine a cut of $V$ into $U$ and $V \setminus U$ that respects $A$. Let $(u, v)$ be a light edge crossing the cut. Then the edge $(u, v)$ is *safe* for $A$ — we can add it to $A$, and the set $A \cup \{(u, v)\}$ is still part of some MST.

This is not hard to prove, so we will leave it to other places; we use the same cut-and-paste proof as for the cut property, but with a few more details. The difference in setup now lets us do cumulative addition of safe edges.

**Corollary 9.14**

Suppose we have the same setup — a graph $G$ and a subset $A \subseteq E$ included in some MST. Suppose we have a connected component $C = (V_C, E_C)$ in the subgraph $G_A = (V, A)$. Then if $(u, v)$ is a light edge connecting $C$ to some other component in $G_A(V, A)$, then $(u, v)$ is safe for $A$.

*Proof.* Take the cut $(V_C, V \setminus V_C)$ — then this cut respects $A$, while $(u, v)$ is safe for $A$. □

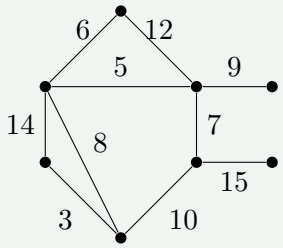
## §9.6 Prim's Algorithm

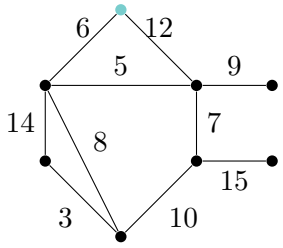
In Kruskal's algorithm, we added *edges* one at a time. Now we'll look at another algorithm that focuses on *vertices*, built around this corollary.

**Algorithm 9.15 —** Take our graph, and select a vertex $r$ to start. We start with an empty tree, and we put $r$ into it. Then until we're done, we select a light edge that connects our growing tree to an isolated vertex — call that edge $(u, v)$ — and we add that isolated vertex to the growing tree.

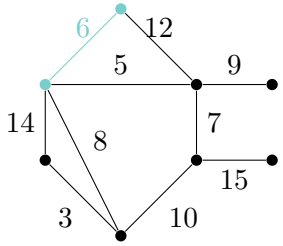
**Example 9.16**

Consider our example graph:

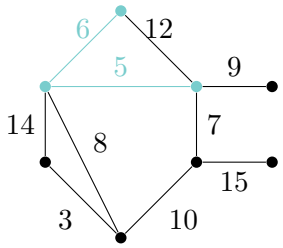



Suppose we start with the top vertex as $r$. Then we select a light edge connecting $r$ to an isolated vertex. There are two choices, 6 and 12; so we choose the light one, which is 6.




Now our options are 12, 5, 8, and 14; the lightest is 5.




In Kruskal's algorithm, we took edges in order — we ahd a growing component somewhere, and another growing component somewhere else. Meanwhile, Prim's algorithm just grows *one* component, by adding an edge that connects a vertex every time. These are quite different.

We can prove correctness by taking the cut with our growing tree on one side, and all the remaining vertices on the other.

Now how do we implement this? One way is by using a *min-priority queue*:

- We can insert an element $x$ intoa set of elements $S$.
- We can find the minimum element of $S$, and we can also extract it out of $S$.

- We can decrease the value of an element $x$'s key to the new value $k$.

In this case, our priority queue tracks vertices that haven't already been added. The key is the lightest-weight edge from that vertex to the growing component. Then we pull out the vertex we want to add, and update the keys if necessary — while the queue $Q$ is nonempty:

- We extract the minimum $u$ from $Q$, and we add $u$ to the list (along with its parent $u.\pi$ that the lightest edge corresponds to).
- Now for each $v$ adjacent to $u$, if $v$ is still in the queue, we check whether the new edge $uv$ is lighter than the previous cost it would have taken to add $v$. If so, we update its parent $v.\pi$ to $u$, and its key to $w(u, v)$.

We have to first initialize, and then we loop through all the vertices. For each vertex, we also have an inner loop, where we visit all its edges. Each edge is only executed once — so the runtime depends on the implementation of the min-priority queue.

> **Exercise 9.17.** Two MST properties you may try to prove:
>
> - A graph with unique weights that has a MST will have a unique MST.
> - If a graph has a cycle with a (strictly) heaviest edge, then that edge will not be in any MST.

# §10 October 6, 2022 — Maximum Flow I

We'll stick with the topic of graphs, but instead of looking at trees, we'll now look at flows through directed graphs. There is a lot of machinery that goes into this topic, so today we will introduce a lot of that machinery, and when we pick up this topic again, we'll quickly review it and go on to the rest of the topic.

We'll look at flows inside graph networks — and especially how to compute a *maximum flow*, the most amount of material we can have flow through the network. We'll see that one way of getting a maximum flow is to start with *some* flow, and visualize things in a way where we can see a route that allows us to increase the flow (which we'll call an *augmenting path*), and then seeing if we can find another, and so on. In the end, we'll state the *max-flow min-cut theorem*.

Maximum flow is really important in infrastructure. When you *build* infrastructure, you want to minimize the cost of laying down the infrastructure, and that's what the minimum spanning tree is for. Meanwhile, maximum flow is for once you have the infrastructure, and want to see how much stuff you can push across it — if you can increase capacity along some routes, which are the bottlenecks you should spend more money on to flow more stuff?

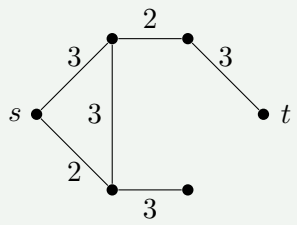
## §10.1 The Setup

Our networks will be *directed graphs* $G = (V, E)$, with some special attributes: one vertex, called $s$, will be the **source**. Material will flow out from the source. Another special vertex, called $t$, will be the **sink**. Material will get delivered to the sink.

The graph also has **edge capacities**, which tell us how much material we can flow over that edge — capacities are given by a function $c\colon E \to \mathbb{R}_{\geq 0}$. It's convenient to have a capacity between all pairs of vertices, whether or not there's a corresponding edge in the graph; so we define the capacity of an edge not in the graph to be 0 (so $c(u, v) = 0$ if $(u, v) \notin E$).

**Example 10.1**

Take the following graph:




We can imagine the capacities as the number of lanes in a directed highway. We can look at the top set of three edges; even though the capacities are 3 on the left and right, there's a bottleneck of 2 in the middle, so we can really only push 2 lanes across the top set of three edges.

**Definition 10.2.** In a valid **flow**, the flow across every edge should be at most the capacity of that edge, and the flow *into* every vertex (other than the source and sink) equals the flow out.

It's also true that the amount of flow out of the source equals the amount of flow of the sink. This is an expression of conservation — lanes of traffic can't be created or destroyed, but are only moved around.

**Definition 10.3.** The total flow of the network is the amount coming out of the source.

In this example, the flow is 3.

**Definition 10.4.** We call a flow like this a *real flow* or *gross flow*. We can formalize this by writing $g(u, v)$ as the amount of flow across $(u, v)$. We should have the properties that:

- $0 \leq g(u, v) \leq c(u, v)$ for all $(u, v) \in E$.
- For all vertices $v$ other than $s$ and $t$, $\sum_u g(u, v) = \sum_u g(v, u)$.

When we're faced with a problem, we'd like to understand some mathematical properties. To do so, we'll get past this definition of *gross flow*, and instead look at *net flow*. This makes a lot of the ideas much more naturally.

## §10.2 Net Flow

**Definition 10.5.** The **net flow** is a function $f : V \to V \to \mathbb{R}$ (which can be negative) such that:

- $f(u, v) \leq c(u, v)$ for all $u, v \in V$ — we call this property *feasibility*.
- $\sum_u f(u, v) = 0$ for all $v \neq s, t$ — we call this properrty *flow conservation*.
- We have $f(u, v) = -f(v, u)$. This is called *skew symmetry*.

So in our graph of real flow, we have a flow of 1 in one direction; if it were instead a net flow graph, we'd also have $-1$ going in the opposite direction. When we do that, the second equation automatically includes the outgoing and ingoing flows — every outgoing edge is there, and every incoming edge has an outgoing partner with negative flow.

Note that we've dropped the condition that $f(u, v) \geq 0$. So how do we ensure that the real parts of the flow are nonnegative? What happens is that the capacity forwards is 3, but the capacity coming back is 0 — there's a $-1$ in there, but the capacity itself is 0. That 0 will ensure that the forward flow can't be negative.

Note that skew symmetry implies that $f(u,u) = 0$ (so we don't have any self-loops). Likewise, $f(u,v)$ is the net flow from $u$ to $v$ — if it's negative, that's because in the gross flow graph, there's a flow of $|f(u,v)|$ in the opposite direction (from $v$ to $u$).

Note that we'll have $f(u,v) = 0$ if $(u,v)$ and $(v,u)$ are both not in $E$. Also, for every edge in the gross flow graph, we've added its opposite; so we can't have a gross flow graph that has an edge and its opposite. THis limits us a little bit — we can't have two-way highways, and can only have one-way highways. It's possible to fix this by simply adding a phantom vertex — we keep the forwards edge where it is, and send what would be backwards through the phantom vertex. We can use this to convert any gross flow graph that violates this condition to one we can use. (We don't allow repeated edges.)

**Definition 10.6.** The **value** of a flow is $|f| = \sum_{v \in V} f(s,v)$.

Some of these terms may be positive, and some may be negative. To avoid writing out all the summations, we will write this as $f(s,V)$.

**Problem 10.7** (Maximum Flow Problem)**.** Given a network expressed as a graph $G = (V,E)$ with $s$, $t$, and $c$, find a flow in this graph of maximum value.

It's useful to think about some legitimate flows.

**Example 10.8**

One acceptable flow is to have no flow, i.e., $f = 0$.

**Example 10.9**

Another legitimate flow is to have a vertex $u$, and have flow leave $u$, go through a series of vertices, and come back to $u$ (with the same amount of flow in each edge), with the condition that the flow in each edge can't exceed the capacity.

**Example 10.10**

If we have the vertices $s$ and $t$, a flow that leaves $s$ and enters $t$ is also an example of flow (with the same value of flow along each edge, not exceeding the capacity of any of the edges).

Of course, there are more interesting examples. But what's useful about this set — flow cycles and $s-t$ paths — is that we can take *any* valid flow and decompose it into a linear combination of flow cycles and $s-t$ paths. The $s-t$ paths contribute to the overall flow, leaving $s$ and ending up in $t$ (as long as the flow value is greater than 0). Meanwhile, cycles contribute 0 to the overall flow — even if $s$ and $t$ were in the cycle (since anything leaving $s$ and going to $t$ would also leave $t$ and go to $s$). So if we want to create a maximum flow, we'd like as many $s-t$ paths as we can, in some sense.

We can say this more formally:

**Definition 10.11.** Let $\text{supp}_f(G)$ (called the *support*) be the subgraph of $G$ of edges $(u,v)$ with $f(u,v) > 0$ — that is, a graph of the gross flow of $f$.

**Lemma 10.12** (Flow Decomposition Lemma)

For any flow $f$ with $|f| \geq 0$, $\text{supp}(G)$ can be decomposed into a collection of $s-t$ paths and flow cycles.

*Proof sketch.* If $|f| > 0$, then there's at least one outgoing edge from the source with positive flow. Suppose we follow that edge. Then by conservation, if this is a valid flow, then it must go someplace — so it has to continue onwards. We can do some sort of network search to follow the train of conservation. Then one of two things happens: either it comes back to some vertex, or it has to end up in $t$. Either way, we can take the minimum value along either the cycle or the $s-t$ path, and subtract it out from all the edges in the cycle or the path. That has to zero the flow in at least one edge.

Now we have a smaller graph left — it's missing one edge. We can repeatedly do this to pull out all the components until there's nothing left — and the only things we'll pull out are flow cycles and $s-t$ paths. □

This leads to two thoughts:

**Exercise 10.13.** Show that the idea of flow decomposition leads to the relationship that the number of $s-t$ paths and the number of flow cycles in the decomposition add up to at most the number of edges of nonzero capacity.

**Exercise 10.14.** If $|f| > 0$, then there exists at least one $s-t$ path.

## §10.3 Describing a Maximum Flow

Now let's think about how we might work towards finding a maximum flow. We said that $f$ was any flow; so let's say that $f^*$ is a maximum flow, and let $F^* = |f^*|$ be its value.

**Question 10.15.** Is $F^* > 0$?

Let $G^+ \subseteq G$ be a subgraph with edges of positive capacity. Then the existence of a $s-t$ path in $G^+$ implies we have capacity going from $s$ to $t$, so we can push flow across it to create its own positive amount. Even if we have a cycle, we could avoid half the edges, and just push flow from $s$ to $t$.

On the other hand, can we show that $F^* = 0$? This introduces the idea of a *cut*: Let $\hat{S}$ be the set of vertices such that there exists a path to those vertices from $s$ in $G^+$ (of course $s \in \hat{S}$). If the maximum flow is 0, then we must have $t \notin \hat{S}$ — instead, $t \in V \setminus \hat{S}$.

So there may be a lot of vertices that are reachable from $\hat{S}$, but if there's a break in $G^+$, then there's no capacity across this gap, and that's proof that there's no $s-t$ paths.

This separation into two sets $\hat{S}$ and $V \setminus \hat{S}$ is called a **cut**. A $s-t$ cut has $s$ on one side of the cut, and $t$ on the other side.

A $s-t$ cut that respects the edges of $G^+$ is certification that the maximum flow is 0.

We can say this formally:

**Definition 10.16.** A $s-t$ cut is a cut $S, V \setminus S$ such that $s \in S$ and $t \in V \setminus S$.

**Definition 10.17.** The **capacity** of a cut is

$$c(S) = c(S, V \setminus S) = \sum_{u \in S} \sum_{v \in V \setminus S} c(u, v).$$

We've been talking about the max-flow problem, but there's another closely related one:

**Problem 10.18** (The $s-t$ cut problem)**.** Given $G = (V, E, s, t, c)$, find a $s-t$ cut of minimum capacity.

We'll see that there's a connection between these two problems.

## §10.4 Maximum Flow and Minimum Cuts

Let $S^*$ be a minimum capacity $s-t$ cut.

**Question 10.19.** What is the flow across the cut?

In implicit notation, we write $f(S) = f(S, V \setminus S)$; in explicit notation, this is

$$f(S) = \sum_{u \in S} \sum_{v \in V \setminus S} f(u, v).$$

By feasibility, $f(u, v) \leq c(u, v)$; therefore

$$f(S) = \sum_{u \in S} \sum_{v \in V \setminus S} f(u, v) \leq \sum_{u \in S} \sum_{v \in V \setminus S} c(u, v) = c(S).$$

So then what we've shown is the following:

**Proposition 10.20**

The flow across any cut is bounded by the capacity of that cut. In symbols, for any $s-t$ cut $S$, we have $f(S) \leq c(S)$.

**Proposition 10.21**

For some valid flow $f$ and any two $s-t$ cuts $S$ and $S'$, we have $f(S) = f(S')$.

Intuitively, imagine we have a $s-t$ path, with flow across it — for example, one with flow 3, 3, 3. We can cut it in any of three places; all have the same flow. Then this claim states that this is true for *any* complicated flow; this is unsurprising, because of the path decomposition lemma.

Recall that we defined $|f| = f(\{s\})$. But by the above proposition, this must equal $f(S)$ for *any* $s-t$ cut. In particular, this means that the maximum flow is bounded by the minimum $s-t$ cut — if $F^*$ is the maximum flow value, then we must have

$$F^* = |f^*| = f^*(S) \leq c(S)$$

for any cut $S$, and we may as well choose the cut with minimum capacity — so then

$$F^* \leq c(S^*).$$

## §10.5 Ideas for an Algorithm

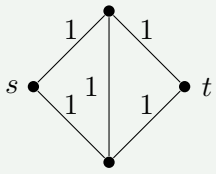
These are the foundational mathematical tools we'll use to study flow networks. Now let's talk directly about how we might get a maximum flow.

One idea is to repeatedly search for $s-t$ paths in a residual flow network — we can start with some graph, find a $s-t$ path, and remove it from the capacity network, and now look for another flow. We keep adding flow and reducing capacity, until we have no capacity left.

Let's think about what that looks like in practice. The problem with that idea is that it's easy to draw extremely simple graphs that look like dead ends:

**Example 10.22**

Take the graph:




It would be bad if we chose some path, such as $s12t$ — then we get stuck. It would be nice to be able to say that we have a capacity of 1 to *undo* that — we'd like to be able to choose the second zigzag flow with 1, $-1$, 1 in order to cancel out the flow in the middle.

So the problem is recognizing that we can take 1 away — and that's motivation for what follows.

In the above example, we can see that the maximum really is 2 — we can make a cut with all three vertices on one side and $t$ on the other, and the capacity of that cut is 2 — so the maximum flow cannot be greater than 2.

## §10.6 Residual Networks

The *residual network* sort of expresses how much capacity we have left — we'll build up a flow towards the maximum flow, and keep track of how much unused capacity we have, looking for $s-t$ paths that we can find there.

We want to be able to represent that we have capacity to drop $+1$ back to 0.

**Definition 10.23.** If $f$ is a net flow on $G=(V,E)$, then the **residual network** $G_f(V,E_f)$ is a weighted, directed graph as follows:

- For every $(u,v)\in E$ (in our original network), we define a **flow capacity** $c_f(u,v)=c(u,v)-f(u,v)$. If this is greater than 0, then we add $(u,v)$ to $E_f$. (So if we still have free edges, we add them to the new graph.)
- Meanwhile, if $c_f(v,u)=c(v,u)-f(v,u)>0$, then we put the backwards edge $(v,u)$ into $E_f$.

For example, let's suppose we take the bad zigzag path in our exapmle. Now if we look at the vertical directed edge $u\to v$, we can see that $c(u,v)-f(u,v)$ is not positive, because it's 0 — so it's not in the residual graph, telling us that we can't push any more flow upwards. But we *can* push flow downwards. The other direction tells us that $0-(-1)>0$ (the flow upwards is 1, so in a net flow graph, the flow downwards is $-1$), so we instead get to introduce the opposite edge — we now have residual capacity 0 going up, but 1 going down because we can push 1 downwards (which is what we need to do in order to push one down).

So the residual network is just the normal graph, except we define the weights as the residual capacities. This depends on teh flow.

Let's take the gross flow graph network we drew initially. Now if we draw the residual graph, we have an edge with 2 going upwards for the 1/3 edge, because we could push 2 more upwards; but we've pushed 1 up, so we also have an edge of 1 going downwrads (because we've added a flow of 1, and now we can remove it if we want to). We can play with every edge — we could push 1 more or remove 2 from it, and that would keep it in the legal range.

The other property here is that we have to be able to create a $s-t$ path in here to have a full augmenting path. If we can't find a way to get from $s$ to $t$ in thi sgraph, then we can't augment the flow.

With that in mind, in this case we *can* augment the flow, by pushing 1 flow — that would be a valid augmenting path. SO then if we start with $G$ and we find $G_f$ and an augmenting path $P$, we can add that back to $G$. And then we can recompute this, and maybe we can find an augmenting path or maybe we can't. If there's a gap then we can't.

### §10.7 Augmenting Paths

**Definition 10.24.** A path from $s$ to $t$ in $G_f$ is an **augmenting path** in $G$ with respect to $f$. THe flow value can be increased along an augmenting path $p$ by the amount $c_f(p) := \min_{(u,v)\in p} c_f(u,v)$.

We'll finally end with teh statement of the max-flow min-cut theorem:

**Theorem 10.25**

Suppose $f$ is a net flow. Then the following are equivalent:

- $|F| = c(S < T)$ for some cut $(S, T)$.
- $f$ is a maximum flow.
- $f$ admits no augmenting paths.

## §11 October 18, 2022 — Max Flow

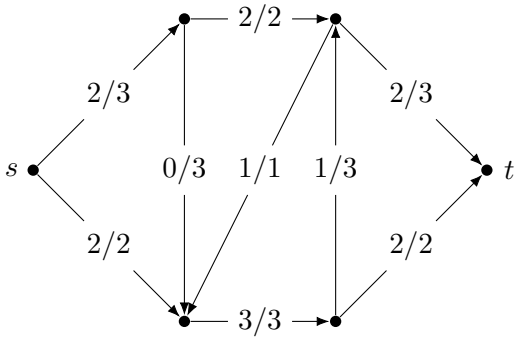
Last time, we introduced *flow networks* — directed graphs in which material can flow. We learned how to decompose flows. We also learned how to see whether the flow a network was carrying could be increased, via a *residual graph* that lets us keep track of unused capacity to produce flow. The idea of an *augmenting path* — a continuous pathway from the source to the sink that has capacity at every link in the chain — can be used to increase flow by pushing flow along one of these augmenting paths. Last class, we ended by stating the max-flow min-cut theorem.

Today we're going to review the highlights, and then prove the max-flow min-cut theorem. Then we'll look at two algorithms for computing max flow. In many problems, capacities are always integers; that leads to some properties we can make use of and prove. Finally, we'll talk about some applications of max flow.

### §11.1 Review of Setup

Last time, we saw that a flow network has a *source* $s$, a set of directed edges, and a *sink* $t$. Every edge has a nonnegative *capacity* associated with it — how much stuff it can carry.

In a valid flow, we run material through each edge, with certain constraints. We'll call $g$ the *gross flow*:



The two constraints are that every edge can carry at most its capacity, and material can't pile up anywhere — at every vertex except the source and sink, we must have the same amount of material coming in and out. These two criteria are called *feasibility* (we don't exceed capacity), and *flow conservation* (the amount of flow entering and leaving each vertex is equal).

Looking at this flow a bit more carefully, we can see that across the top, we send 2, and then 2 more, and then 2 more. This means we could just pick out a flow of 2 that goes across the top from $s$ to $t$. Likewise, looking along the bottom, we see 2, 3, and then 2. We can break the 3 as $2 + 1$, and that gives us another flow of 2 going across the bottom.

Then we're left with a circle of 1, 1, and 1 — that's a cycle of flow value 1.

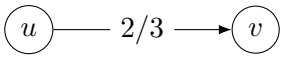
Last time, we discussed that *any* flow can be decomposed as a collection of $s$-$t$ paths and cycles; this is an example.

> **Student Question.** Is the flow in a cycle 0?
>
> It contributes 0 to the flow from $s$ to $t$, but it exists — if we get rid of it we'll have a different flow pattern, but the total flow value is the same.

In our example, the value of the flow is 4, and that turns out to be a maximum flow. But how can we show this?

### §11.1.1 Net Flow

So far, we've drawn a *gross flow* — what pipes look like in the real world. But often, it's more useful to look at the *net flow*. Let's consider a little piece of the gross flow:




In this example, we have $g(u, v) = 2$, $g(v, u) = 0$ — when we defined flow networks we required that we can't have edges in both directions — and $c(u, v) = 3$ and $c(v, u) = 0$.

In this example, to find the *net flow*, we'd still have $c(u, v) = 3$ and $f(u, v) = 2$. But on the opposite edge, we'd define $f(v, u) = -2$. (We still have $c(v, u) = 0$.)

Using this, we can now compute residual flow and look for excess capacity, and find augmenting paths we can use to increase the capacity of an existing flow.

With these net flows, our properties are simplified:

- By definition, $f(u, v) = -f(v, u)$ — this is called *skew symmetry*.
- Feasibility still means that $f(u, v) \leq c(u, v)$. (Unlike in gross flow, we no longer require that flow is nonnegative. But along with the first condition, since the capacity of the reverse edge is 0, the flow across the reverse edge must be nonpositive, and therefore the flow across the actual edge is nonnegative.)
- Flow conservation now becomes that
$$\sum_u f(u, v) = 0$$
for all $v \neq s, t$ — the sum of flow entering every vertex is 0. What's happening here is that in the gross flow, we had to explicitly count the edges pointing inward and add them up, and then subtract the edges pointed outwards (or the other way around). Here that's taken care of automatically — each edge has a forward and reverse version, and so edges out are counted with the opposite sign as edges in. So this automatically takes care of the minus sign for us.

### §11.1.2 Cuts

We defined a *cut* as a partition of $V$ into two sets $S$ containing $s$ and $V \setminus S$ containing $t$ (where the other vertices could be assorted in any way). We found last time that there's a special relationship between the capacity of a cut and the amount of flow that could flow across the cut.

The capacity of a cut is defined as

$$c(S) = c(S, V \setminus S) = \sum_{u \in S} \sum_{v \in V \setminus S} c(u, v).$$

Here we're summing over all edges from the $s$-side of the cut to the $t$-side of the cut.

Last time, we saw the following:

**Lemma 11.1**

For any cut $(S, T)$, we have $|f| = f(S, T) \leq c(S, T)$.

In particular, this means the maximum value of the flow can't be any greater than the minimum capacity of a cut. In fact, as we'll soon see, they are *equal*.

### §11.1.3 Residual Networks

The idea of a residual network is to tell us where our bottleneck paths are, and where we can push more flow across.

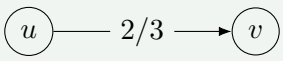
**Definition 11.2.** Let $f$ be a net flow on our graph $G$. Then the *residual network* $G_f$ has all our vertices of the original graph, and a set of edges $E_f$ along with residual capacities

$$c_f(u, v) = c(u, v) - f(u, v),$$

such that we add an edge to $E_f$ if and only if its residual capacity is strictly positive.

**Example 11.3**

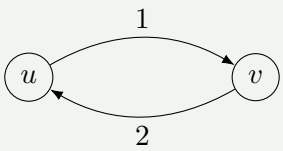
In our small example:




Here the capacity of the edge is 3, but we're only pushing across 2 units of flow. So $c_f(u, v) = 3 - 2 = 1$.

Meanwhile, for the reverse, we have $c_f(v, u) = 0 - (-2) = 2$.

So then our edges in the residual graph are as follows:




What does this mean? In the original graph, we have a one-way pipe from $u$ to $v$ — we can't pump any flow backwards?

That's because we've previously put a flow of 2 in there. So we can undo some or all of that flow of 2, as we augment the flow! That's the key feature that this mathematics achieves for us — it takes all this undoing and writes that it's like sending 2 backwards.

Notice that the residual capacity of an edge and its reverse edge should always add up to the original capacity.

**Student Question.** *Is the capacity of any edge not in the graph* 0*, in both directions?*

**Answer.** Yes.

### §11.1.4 Augmenting Paths

Once we have the residual network, we can look for an augmenting path:

**Definition 11.4.** Any path that goes from $s$ to $t$ in the residual graph $G_f$ is an augmenting path in $G$ with respect to the existing flow $f$.

If we have an augmenting path, then we can increase the flow value $|f|$ along the augmenting path, by its *bottleneck capacity* — the smallest capacity in the residual network along that path. In other words, we define the residual (or bottleneck) capacity of a path as

$$c_f(p) = \min_{(u,v)\in p} c_f(u,v).$$

If we can send 100 units through all but one steps in a path, but only 2 edges through one, then we can only send 2 through the path.

In other words, if we have an augmenting path in our residual graph, then we can take the existing flow, and add a flow along that path of its bottleneck capacity.

## §11.2 The Max-Flow Min-Cut Theorem

**Theorem 11.5**

Suppose $f$ is a net flow. Then the following three statements are equivalent:

(1) $|f| = c(S,T)$ for some cut $(S,T)$.

(2) $f$ is a maximum flow.

(3) $f$ admits no augmenting paths.

We'll now prove this theorem. We want to prove (1) $\iff$ (2) $\iff$ (3). The way we're going to prove this is by showing

$$(1) \implies (2) \implies (3) \implies (1).$$

Once we've got a circle, then any one implies any of the others — for example, to show (3) $\implies$ (2) we can say that (3) $\implies$ (2).

*Proof of* (1) $\implies$ (2). Previously, we've seen that the value of *any* flow must be less than or equal to the capacity of any cut — in other words, $|f| \le c(S,T)$ for any flow $f$ and any cut $(S,T)$. So if we have $|f| = c(S,T)$ for some cut, then *every* flow $f'$ must satisfy $|f'| \le c(S,T) = |f|$ — so we must have the maximum flow. □

Note that we call this the *max-flow min-cut theorem*. We have a maximum flow here, but in the statement, we just have *some* cut. But it's also true that this cut must be the minimum cut, by almost the same logic — if $|f| = c(S,T)$ for some cut $(S,T)$, then $(S,T)$ must be a minimum cut (no other cut can have lower capacity than this one). This is because we must have $|f| \leq c(S',T')$ for *any* cut — if this were violated, then the flow would exceed the capacity constraint of at least one of the edges. And we're told that $|f| = c(S,T)$ for a particular cut $(S,T)$. So again, we then have

$$c(S,T) = |f| \leq c(S',T'),$$

and thus our cut is a minimum cut. (This is the same logic, just turned around and applied to the cuts instead of the flows.)

*Proof that* (2) $\implies$ (3). We want to show that if $f$ is a maximum flow, then we can't find any augmenting paths. We'll instead prove the contrapositive — that $\neg(3) \implies \neg(2)$. Imagine that there *is* an augmenting path for $f$. Then we can find its bottleneck capacity, add that new path to $f$, and get a new flow $f^+$ that has increased flow value. That would violate (2) — so if (3) is violated, that causes (2) to be violated. And therefore if (2) is true, then so is (3). □

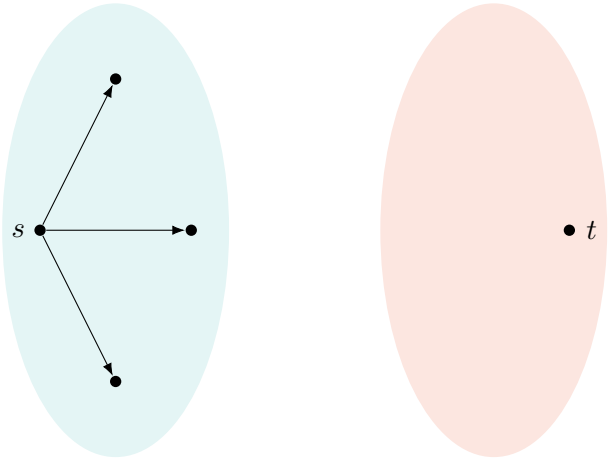
This results directly from the definition of an augmenting path.

*Proof that* (3) $\implies$ (1). We want to show that if $f$ has no augmenting paths, then its flow value is equal to the capacity of some cut. This one is more interesting: imagine we have a source $s$ and a sink $t$, and we look at the residual network for our flow $f$ that admits no augmenting paths.

Now take $S$ to be the set consisting of exactly all the vertices reachable from $S$ in this graph. Of course $s$ is reachable from $s$, so it's in $S$; some other vertices may also be in $S$.

Now we can let $T$ be the set of vertices that are *not* reachable from $S$. Then we know that $t$ is in $T$ — if $t$ were reachable from $s$, then by definition we'd have an augmenting path from $s$ to $t$. (By definition, all edges in the residual graph have *positive* capacity.) There may be more vertices in this set as well.

Then we can draw a cut separating $S$ and $T$.




Then this cut has zero capacity in our residual graph — there's no edges that cross it.

Let's look at a particular edge in the original graph $(u,v)$ that crosses the cut. Then we must have

$$0 = c_f(u,v) = c(u,v) - f(u,v),$$

which means $c(u,v) = f(u,v)$.

But then summing over all $u \in S$ and $v \in T$, we get

$$\sum_{u \in S} \sum_{v \in T} c(u, v) = \sum_{u \in S} \sum_{v \in T} f(u, v),$$

and therefore we have

$$c(S, T) = f(S, T) = |f|,$$

since the flow across any cut is equal to the value of the flow (by flow conservation).

So then this gives us exactly statement (1) — the value of the flow in the real graph is equal to the capacity of our cut in the real graph.

Intuitively, what this says is that if there's no way to get from $s$ to $t$, then there's a cut we can't cross, and all edges on that cut are filled to capacity. □

**Student Question.** *Is $f$ a net flow or a gross flow?*

**Answer.** Net flow — we'll use $g$ to denote gross flow, and we'll do almost all our work with net flow.

## §11.3 Ford–Fulkerson Algorithm

Now let's look at a few algorithms.

**Algorithm 11.6** (Ford–Fullkerson, 1956) **—** First, initialize $f[u, v] = 0$ for all $u$ and $v$. Now while an augmenting path exists in the residual network, we find it and increase the flow by its bottleneck capacity. We keep doing this until we can't find any more augmenting paths.

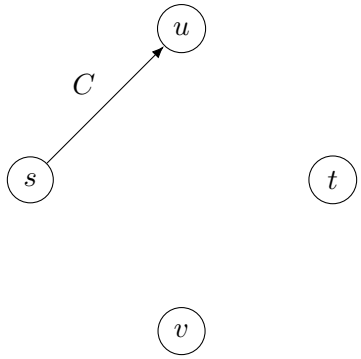
It's easy to see this is correct — we terminate only when there's no augmenting paths, and by (3) this means $f$ must be a maximum flow.

The problem is that this doesn't give us a good way to find augmenting paths — this is done by breadth-first search. If we think about runtime, each iteration or augmentation takes $O(|E|)$ time, since we may need to search all edges. Meanwhile, if we have integral capacities in $[0, C]$, then the number of iterations is at most $|f^*|$ (since all bottleneck capacities will be integers), which is at most $c(\{s\})$. In the worst-case, $s$ may be connected to every vertex in the graph with the maximum capacity — so we could have $c(\{s\}) = VC$. This leads to a worst-case total runtime of $O(EVC)$, which could be very, very large.

**Remark 11.7.** We call this *pseudo-polynomial* because it's not polynomial in the binary encoding of the input. If we run through the edge set, then we'd have an array of the edges, so the input size for the edge set is proportional to the number of edges.

But the same isn't true for the capacity — if we double the capacity, we only add one bit, not twice as many!

There's a typical example that goes along with this. Imagine we have a graph with just four vertices:




Suppose we perform our search, and the augmenting path we come up with is $s \to u \to v \to t$. This is obviously not the best augmenting path, but it's possible it's the one we find. It has a bottleneck capacity of 1. So now we push one unit of flow through that path, and we get the following:

Now the path we might find is $t \to v \to u \to s$ — we might get unlucky again. And this can happen $2C$ times, which is really bad.

## §11.4 Integer Flow

Note that if all capacities are integers, then the above algorithm will *always* produce integers — the bottleneck capacities are integers, so we're only ever adding and subtracting integers.

That leads to the following useful theorem:

**Theorem 11.8**

If we have a flow network such that $c(u, v) \in \mathbb{Z}$ for all edges $(u, v)$, then there exists a maximum flow $f$ such that $|f|$, as well as all values $f(u, v)$, are integers.

Note that it's possible to find a graph that has integer capacities but a non-integer maximum flow — the theorem only tells us that there *exists* an integral maximum flow.

## §11.5 Edmonds–Karp Algorithm

The key observation is that in our bad example, the number of edges in our bad pathway was 3. If we'd chosen a pathway with just 2 edges, we'd be much better off.

Edmonds and Karp were able to generalize that — they showed that if of all the possible augmenting paths, you chose the one that's the shortest unweighted path in the residual graph — the one with the smallest number of edges — then the algorithm would run faster.

**Lemma 11.9** (Edmonds–Karp Lemma)

If we always augment with the shortest path in the unit-weight graph, there are at most $O(VE)$ augmentations in the worst-case.

Note that this no longer depends on the capacity!

The shortest path in the unit-weight graph (the graph where each edge-weight is 1 — we essentially just ignore our edge weights and change them to 1 to find the shortest path, and then put back the weights) can be found by breadth-first search in $O(E)$ time. So then Edmonds–Karp runs in $O(VE^2)$ time.

**Remark 11.10.** The notes describe some faster versions. We aren't responsible for them, but they're included because we might find them interesting.

**Student Question.** *Why does this take $O(VE^2)$ time? What if the shortest path still has bottleneck capacity 1?*

**Answer.** This is because of the lemma, which we haven't proven — there's an upper limit on the number of augmentations we'd have to do, that doesn't depend on $C$.

## §11.6 Applications

If we're a civil engineer trying to pipe stuff around, then max flow is useful. But it's also useful for completely different problems, that can be expressed as a max-flow algorithm!

### §11.6.1 Bipartite Graph Matching

**Definition 11.11.** A **bipartite graph** is a graph with two vertex sets $L$ and $R$, such that all edges connect a vertex in $L$ to a vertex in $R$ (there are no edges within $L$ or within $R$).

**Definition 11.12.** A **matching** of a graph is a set of edges where none of the edges share a vertex.

**Question 11.13.** Given a bipartite graph, what's the maximum number of edges we can take in a matching (i.e. the maximum number of edges crossing over such that no pair shares a vertex)?

This is a useful problem for assigning lawyers to cases, or jobs to computers, or students to internships, or doctors to patients. It might be that the first doctor can only fix what's wrong with patient A or patient C.

Why is this a maximum flow problem? Let's do a *reduction* — we'll take our original graph, and convert it into a flow path. We draw $L$ and $R$, and direct all edges $L \to R$. Then we add a source $s$, and add directed edges from $s$ to each vertex of $L$, with capacity 1; similarly we add a sink $t$, and add directed edges from $t$ to each vertex of $t$. We also set all capacities to 1 (since we're trying to get one match for everyone).

Then we can solve for the max flow with Ford–Fulkerson, and we'll get a max flow that corresponds to the matching we want — all the black edges with flow are part of the matching, and all the ones that don't have flow aren't.

It takes $O(m+n)$ time to alter the graph. Then since the maximum capacity is 1, we can run Ford–Fulkerson in $O(mn)$. Then it takes $O(m+n)$ time to figure out what our answer is (reversing the modifications). So this takes $O(mn)$ time.

# §12 October 20, 2022 — Linear Programming

## §12.1 Introduction — A Toy Problem

**Example 12.1**

Suppose that at your favorite arcade, you have two claw cranes $A$ and $B$, which pick up toys. If you spend $x$ on $A$, you get stuff worth $x/3$. If you spend $y$ on $B$, then you get stuff worth $y/2$.

You want to spend 60 on these two machines. However, you must spend at least as much on $A$ as you do on $B$. Also, if you don't spend all the money, your parents take back the remains.

How do you get the most out of these two machines?

Let's formulate this in the following way. Our *objective* is to maximize $x/3+y/2$, subject to teh constraints:

- $x \geq y$ (which we can rearrange to $-x+y \leq 0$.)
- $x+y \leq 60$.
- $x \geq 0$ and $y \geq 0$. (These are obvious, but we'll see soon why we want to write these down explicitly.)

This is called a **linear program**. There's a lot of linear things about it — the **objective function** $x/3+y/2$ is linear in $x$ and $y$, adn the constraints are also linaer.

If we have some such thing, how do we go about solving it?

Adding the constraints, we get $2y \leq 60$, so $y \leq 30$. Intuitively, we get more returns from $B$ than we do from $A$, so we want to spend more of our money on $B$. We're constrained to say we can't spend more than 30, so we spend all 30 on $B$, and the remaining on $A$ (because if we don't spend it, we get no returns on it.) Intuitively, we push $y$ to be as large as possible, and then we push $x$ to be as large as possible given the new constraints. This gives us an answer of 25.

But what if the constraints weren't so simple? What if we couldn't just add up our equations to make things vanish, or if there were more constraints? We'd somehow have to pick them in a smart way so that we could elmiminate variables.

So now let's try to use a bigger hammer that generalizes — a graphical, or geometric, approach.

Let's start with a blank $xy$ graph. Now we want to see what our constraints tell us — so let's graph them. First let's graph $-x+y \leq 0$. If this were an equality, it'd be easy to graph — we find two points on the line, and draw the line. If we have an inequality, it also includes more points — if we think about it a little, it's one of the two sides of the plane. So we draw the line $-x+y=0$ (which is $x=y$), and now it's easy to tell which half we want.

Now let's add our next line, $x+y \leq 60$. Again, the line is $x+y=60$, and the origin has $x+y=0 \leq 60$, so we want to take the bottom half.

This gives us a triangle, which is where our solutions lie — that's the space that respects all our constraints. So another way of loking at this problem is that we want to maximize this function, *over that triangle*.

The point at the top is $(30,30)$; the other points are the origin and $(60,0)$.

Now $x/3+y/2$ is also a linear function, so let's draw the line $x/3+y/2=$ some value. Let's starta t 0; that gives us a line. It passes through the origin.

And now the question is, what happens as this line moves? Every point on that line has $x/3+y/2=0$. If we move the line out, the value changes, but all the points on that line have the same $x/3+y/2$ value. As that happens, the line moves parallelly, and as we go up the value increases, and as we go down the value decreases.

Now it's very easy to conclude what the optimal solution is — we just see how far we can push the line so that it stays in our region. And that's exactly at $x/3 + y/2 = 25$ — if we push it further, then we move above the feasible region. We could go lower, but then our objective decreases.

This is a complete proof that in fact, $(30, 30)$ is the actual solution — we've looked at the entire solution space and optimized the objective there.

Now let's go back and think about it more — we'll milk this toy problem to its maximum.

Note tha twhen we add up constraints, we're not generating new information — when we added them to get $y \leq 30$, we're not generating new information, since it's implied by our two constraints. But that was powerful, since we could think of one variable instead of two.

We can view $-x + y \leq 0$ and $x + y \leq 60$ as telling us the maxima of different functions. FOr example, the optimum of $-x + y$ is 0, the optimum $x + y$ is 60. So we can view these as themselves being maximization tools.

ANd now we can say that if we were to take a combination of these, we'd get the maximum of another function — we took one combination by adding them up to get the maximum of $y$. BUt we could take a different combination, and that would give us newer and newer functions on the left and corresponding maxima on the right.

SO let's say we scale $x + y \leq 0$ by $u$, and $x + y \leq 60$ by $v$. Then what we're saying is

$$(-u + v)x + (u + v)y \leq 60v.$$

Now the question is, can we use this to maximize $x/3 + y/2$? Suppose $-u + v$ turned out to be $1/3$, and $u + v$ turned out to be $1/2$. Then we'd get the answer to our problem — we'd get $x/3 + y/2$ is at most something, which is all we need.

## §12.2 Duality

There's a dual tooy problem — we can maximize $x/3 + y/2$ if we had something else that's true, — we have

$$\frac{x}{3} + \frac{y}{2} \leq (-u + v)x + (u + v)y \leq 60.$$

So then we wantt he *best* upper bound from this, subject to the constraints

$$-u + v \geq \frac{1}{3}$$

and

$$u + v \geq \frac{1}{2},$$

and $u, v \geq 0$.

As long as $u$ and $v$ satisfy these constraints, we get *some* upper boun. For example, we can say

$$\frac{x}{3} + \frac{y}{2} \leq x + y \leq 60$$

(since $x$ and $y$ are nonnegative) — this is why the first set of constraints make sense (that $-u + v$ and $u + v$ just have to be at least the thing). The reason we need $u, v \geq 0$ is becaus eotherwise multiplying our ineqalities by negative numbers flips their direction.

So now we have an alternative problem, of trying to linearly combine constraints from our original problem to find an upper bound on our objective funcction.

This is by no means necessary — it is not necessary that we should be able to do this. But it is *sufficient* — if we are able to do this, we get an answer. But if we coudln't, that doesn't mean the problem is unsolvable; just that the approach doesn't work.

Now we have a new problem, and let's graph *this* one. We're now trying to *minize* our upper bound — if you're finding an upper bound, the lowest one is what counts.

The first constraint is that $-u + v \leq 1/3$; that's the space above some line.

Note that the third approach is sort of a general verison of the first approach — you can formulate it like this, and maybe this, for some reason, is easier to solve.

> **Student Question.** Why do we want to minimize $60v$?
>
> WHen we add these up, we get $(-u+v)x+u+v)u \leq 60v$. We're trying to use this as an upper bound for $x/3 + y/2$. We can already say that $x/3 + y/2 \leq 60$, but can it attain 60 at all? Maybe it can only attain 40, or 30. So we want to get this bound lower — higher bounds are not useufl because you may never be able to achieve them.

Now we have a triangle on the top, boxed off to teh left. This is actually different, qualitatively, than the previous one. There we had a bounded region — it fit inside a picture. This doesn't — it goes all the way to the top. But still, that's our space.

And now we can plot what our objective function looks like, which is $60v$. Running it through different values of $60v$, we see that as it goes up, the value increases. As it goes down, the value decreases. We're trying to minimize, so we want to keep the line as low as possible, but within the feasible region. And that happens right at the tip of our thing — and that point turns out to be $(1/12, 5/12)$.

And in fat, if you do cscale your constraints by $1/12$ and $5/12$, then you exactly get $x/3 + y/2 \leq 25$.

So in stead of our first approach adding them up directly, we could scale them and add them up.

Note that there's a slight caveat — even though you can do this, that doesn't mean it's tight. We still haven't shown that it *attains* 25 — we still have to show that there is a solution.

These $1/12$ and $5/12$ are the 'best multipliers' — they give you your bound in one shot. And now we can go back and say, well, can I flip this on its head? The answer is yes.

Let's suppose that $-u + v \geq 1/3$ and $u + v \geq 1/2$ and maximize $60v$ was our original constraints. Then we could try scaling the first by $x$ and the other by $y$; then our constraints would reverse, and we'd be trying to solve the exact same problem.

So the dual of the dual is the original problem. We call these the **primal** and **dual**, but they are not actually ordered.

Now we have solved the toy problem — the answer is 30 on each for 25.

Now let's go further — we want to generalize.

There was nothing magical about these numbers. So we could reword everything with variables:

> **Question 12.2.** We want to maximize $c_1x_1 + c_2x_2$ subject to the constraints $A_{11}x_1 + A_{12}x_2 \leq b_1$ and ...

What if we add more constraints? Let's say now that we have to spend at least 40 on $A$.

So then we add a constraint that $x \geq 40$, which we rewrite as $-x \leq -40$. Now we can see that this introduces an orange line, and we want to be to the right of the orange line.

Earlier we hit our objective function at 30. But now $(30, 30)$ is left of teh orange line, so it's no longer part of our region. SO the feasible reason shrank — we have a smaller triangle inside the big triangle, and we're

only tehre now. So we actually can't get further than $(40, 20)$ — the moment we do, we cross the feasible region. So our new optimum is $(40, 20)$.

If we again go back, we can argue in many differentw ays why thsi makes sense. We want to make $y$ as big as we can, but if $-x \leq -40$ then $y \leq 20$, since $x + y \leq 60$. So we're still taking $y$ to be as large as we can — which happens t obe 20.

What happens to the dual? You get a new variable $w$ that tells you how much you want to scale our last equation $-x \leq -40$. Then what we want to do is minimize $60v - 40w$, where our equations now become

$$-u + v - w \geq \frac{1}{3},$$

and require $w \geq 0$ as well.

If we tried to plot this, we'd get a 3D plot. This is where visualization starts to get scary.

Now our linear constraints in 3 dimensions are planes. If we plot our two planes, then our feasible region will lie in one of the four parts. The planes will intersect along a line, but only one point along that line will be important for us — which is $(0, 1/2, 1/6)$. The reason is if we now try to plot the objective function and move it through teh space, it'll intersect those two planes at exactly one point. So now we have 3D space and we're moving a plane thorugh it and trying to see where the feasible region attains its minimum, and that happens right there.

This solution turns out to be $(0, 1/2, 1/6)$. There is something interesting that happened — in finding the bound, we ignored the first constraint (by multiplying it by 0). That's important.

> **Question 12.3.** Let's extend to more constraints and more variables. We want to maximize $c^T x$ subject to the constraints $Ax \leq b$ and $x \geq 0$.
>
> In the dual, we want to minimize $b^T y$ subject to teh constraints $A^T y \geq c$ and nonnegativity $y \geq 0$.

We use $x^*$ and $y^*$ to denote the optimal solutions (primal and dual optimals).

> **Question 12.4.** Is $c^\intercal x^* = b^\intercal y^*$?

That occurred in our example.

Now we leave chapter 1 on a cliffhanger.

## §12.3 Standard Form

We've so far been talking about toy problems, but we can generalize this to 'any problem':

> **Question 12.5.** We want to maximize $c^\intercal x$ subject to the constraints $Ax \leq b$ and $x \geq 0$.
>
> In the dual, we want to minimize $b^\intercal y$ subject to the constraints $A^t y \geq c$ and $y \geq 0$.

Note that with this, we can handle basically any linear problem. IF we wanted the opposite inequality, we could just negate them. If we wanted equality, then we could add two-way constraints.

Sometimes we'll want strict inequalities. But that's beyond the scope (and the answer may not exist), so we will treat them at random.

What if we want decision variables over different domains (right now we're saying they're nonnegative).

If we want to take some part of the space — say $x \geq -5$ — then you can look at $x + 5$, which is at least 0. So you can just define a new variable as a shifted $x$.

The other issue one could ask for is — we're essentially taking one side of a line. What if we had part s— $x$ could be between 5 and 10, or between 15 and 20? Then we could split it up into multiple problems.

What about unrestricted variables? We can write them as a difference of two variables that are nonnegative.

There is a ocnnection between equalities and free variables that we will come to later.

So we can actually capture a rich class of problems with this. And we will call it **standard form**.

> **Exercise 12.6.** What is the complexity of converting a LP to standard form?
>
> Are the two standard forms different? (No.)
>
> Can you think of other standard forms that are equally expressive?

## §12.4 Feasibility

We say any solution to teh problem is some setting for the decision variables, which we denote with $\tilde{x}$ and $\tilde{y}$.

If it satisfies all our constraints — $Ax \leq b$ and $x \geq 0$ — we call it feasible (we should have tildes here). The **feasible region** is the set of all feasible points. A feasible LP (linear program) is one that has a nonempty feasible region; an infeasible one is one where there's none (there are no points that satisfy all your constraints).

> **Example 12.7**
>
> In our original toy, the LP isi feasible — its feasible region is a triangle. The points $(20, 50)$, $(50, 30)$, $(10, -10)$ are infeasible; points inside the triangel like $(20, 10)$ are feasible. Any point in the plane is some solution.

> **Example 12.8**
>
> Having a downward diagonal line, a downward upwards line, and then a horizontal line with above, this can have no solutison; that's infeasible.

## §12.5 Optimality

> **Definition 12.9.** AN optimal solution is one such that $c^T x^* \geq c^T \tilde{x}$ for rall feasible $x$.

We say that the feasible region is bounded if it fits inside a ball, and unbounded otheriwse.

A bounded LP is one where all feasible solutions have a bounded obvjective value. An unbounded LP isi one where not. So if you try to optimize an unbounded LP, you can't, because any solution you take, I'll show you another one that beats it. (So existing $M$ such that $c^\intercal x \leq M$ or not.)

As convention, bounded and unbounded LP's are a classification of feasible ones. (We do not refer to infeasible LPs as bounded or unbounded — these ar ethree disjiont cases.)

Going back to our second thing, it's a bounded feasible LP because we found the optimum, but the feasible region is unbounded — this is certainly possible.

If we change the objective function value to be optimizing for $-u$, then because hte region extends on top then we'd have an unbounde d LP instead.

**Exercise 12.10.** A bounded feasible region implies a bounded LP, which implies there exists an optimal solution. (You should convince yourself of this, though you need actual math to prove it.)

Contrapositively, an unbounded LP implies an unbounded feasible region.

## §12.6 Visualize

**Theorem 12.11** (Corner Point Theorem)

THe optima are corner points.

In higher dimensions, we have hyperplanes, and a bunch of half-spaces. We're looking at where all the half-spaces intersect; that is called a **convex polytope** (a weird object in $n$-space). So we have this weird object in $n$-space, and we're moving a plane across that space. The optimum *will* be at a corner. It could be a face or a line or something larger, giving you many optimal solutions, but they all have to lie at the edge of this thing.

The first time people solved this, it was by hopping across corner points. But in $n$-space the number of corner points is exponential in $n$. It was an open question as to whether there exists a polymoial time algorithm, and it was shown later that it *is* polynomial time, with an exponent like 6 or 8. Now we have interior point methods, where instead of hooppoing on the outside we hop on the insid (move inside to find the right cornr), and htose are the best exponents we now.

We are now not going to discuss algorithms for solving lienar programming; the takeaway is tha tit can be solved.

## §12.7 Duality

We left part 1 with a cliffhanger — are the optimal solutions to the primal and dual always equal?

Recall the example where we wanted to find the lowest upper bound, and we formainzed it using the dual and wrote down constraints.

This objective function fothe dual, which at that point was $60v$, was upper bounding the objective function. This should tell you that if you've found any feasible solurtion here for our originla, it will be larger than what we would get here (it can't be smaller) — that's how w edesigned it. The point was to get larger values and keep shrinking it down until we hit the right bound.

This is called **weak duality** — if you have *any* feasible solutions to teh primal and dual, then $c^\intercal \tilde{x} \leq b^\intercal \tilde{y}$. In math, this is because

$$c^\intercal \tilde{x} = \tilde{x}^\intercal c \leq x^\intercal A y \leq b^\intercal y,$$

since $c \leq A^\intercal y$. (Note that there are may constraints, and each is a $\geq$ — we are comparing our vectors pointwise. If you write this out fullly, then if you for instance took $x^\intercal c$ you could write this out as some summation of $x_j c_j$, and then you'd note that $c_j$ is at most some big thing.

A way to think of this is that the dual bounds the primal, which is not surprising because that's what we are trying to do.

The reason we call this 'weak' is it doesn't answer our question. But it gives us a few good observations — if the primal is unbounded (it's feasible but the objective function can't be optimized), then the dual has to be infeasible. That's becaus ie the primal is unbounded, then there are feasible solutions $\widetilde{x}$ where this goes to infinity, so there can't be any $\widetilde{y}$.

Flipping, if the dual is unbounded then the other thing is infeasible. THis means we obviously can't have (unbounded, bounded) or (unbounded, unbounded).

It is psosible that both the primal and dual are infeasible; you can construct trivial examples to show this.

But that leaves us with three question marks.

## §12.8 Rabbit out of Hat

**Theorem 12.12** (Strong Duality)
If the primal is bounded, then so is the dual, and in fact $c^\intercal x^* = b^\intercal y^*$.

This is magic and we will talk about this more in a second.

So to fill up the table, if the primal is bounded, then the dual is bounded, so it's obviously not infeasible. (The table has 'yes' if possible and 'no' if there is no such pair. You can find primal dual pairs that are both bounded, you can find cases where one is unbounded and the other is infeasible, or where both are infeasible. But no other type exists.)

What this says is — our initial approach of adding the equations up, which we generalized, in fact suffices to answer every problem. We can always optimize the primal by optimizing a liena rcombination of its constraints.

So there is this magical thing that optimization in teh linear case turns otu to just be tryign to ombine your constraints. ANd that's really teh heart of linear programming — that this is true,.

## §12.9 Why Care about duality?

In our example, we wrote something that's about the same size. But in certian cases, the dual may be easier to solve. Maybe it has fewer variables — the runtime of our algorithm is mostly based on the number of variables ($L$, which is the problem size encoding the constraints, is linear; but the number of variables grows superlinearly and its conjectured you can't make it linear unless matrix multplication becomes super fast).

So fewr varibles can mean fast er runtime. You amy also have *easier* constraints — andit mayy also have special structure inside. (This is debatable because one looks the same as the other, but it does matter whether it's row-wise or column-wise.)

Strong duality lets us find the optimal objective value. The quesiton we did not answer is how to get the optimal *solution*? Getting the optimal objective function tell sus how to combine the constraints, and that vives the answer. But sometimes the solution is important (it's not important if you tell me 'you an make 25 dollars but I won't tell you how').

But it turns out strong duality tells us more.

Intuitively, the dual solution for $(u, v, w)$ tells us how to scale our constraints to get here. But sometimes, these multipliers may be 0 — we had one where $u = 0$. So it tells us that we ignore that constraint, and just use these two.

In our situation, $-x + y \leq 0$ is not useful information — because our second two things imply not only is $x$ at least $y$, but it's at least 20 more than $y$. So it is *loose* (not tight) — it is not contributing at all to the objective value function. (Our constraints may imply that exactly or somethign even stronger.)

So what this says is, if the dual says I need to select a constraint, then that has to be tight — you need to have equality there. If I'm selecting $x + y \leq 60$ then in the optimal I need to set $x + y = 60$. Equivalently, if this equation has 'slack' (like $-x + y \leq 0$) then the dual will ignore itl because this constraint doesn't give you the right bound — it gives you weaker bounds.

We call this **complementary slack** — when our coefficient is 0 that means something is loose on the left, and vice versa.

In our example, because $(u, v, w) = (0, 1/2, 1/6)$, this says I'm only going t ouse the last two constraints. So those are rtight in teh optimal for the prima.

Looking at our thing, we can see that teh optimal solution $(40, 20)$ is at theintersection of (2) and (3) — and if it's on the line, then we know we have equaltiy. WE know it's at $a$ corner point, but in fact we're at *that* corner point — the corner point of (1) is irrelevant.

In our situation, if we dropped (1) out of teh picture, the feasible region would not change — it is still that bottom triangle. It is not at all counting towards that .

Wha tif the first constraint passed through the point as well? This is left as an exercise. (For example, fi we had that constraint be $-x + y \leq -20$, so that the corner point includes all 3.)

Now not only does the dual solution tell us how to combine, it tells us in fact which constraints are the ones that give us our optimal. So now we kow a way to get the dual solution — we set the stuff to equalities, and we get an answer.

So we can get the solution for teh primal from the dual, and vice versal. You can alos use this to guess soltuions — guess a solution to (1), see which constraints are tight, and look what happens to teh dua. If you have this duality, then the solution will be optimal. And if you find a pair where there's slack in both cases, that solution si not optimal.

### §12.10 Applications

There's a lot of applications. For example, max-flow min-cut is an example of this form — max-flow is of this form and its dual is min-cut, which is why max-flow equals min-cut.

We will use this in game theory next week — Nahs equliibria comes out of this.

Mroe intrestingly this siused in approximation algorithms. We can imagine writing out manh problems in this form with difrerent restrictions — they may not e linear. But the other common thing — n graph problems – is that sometims $x$ and $y$ have to be integers. But you can relax the problem to say actulaly I'll accept fractional solutions, and figure a way to translate to integer soltuions. That gies you approximation algorithms diretly.

## §13 October 25, 2022 — Game Theory

### §13.1 Introduction

**Example 13.1** (Game of Chicken)

Two drivers are headed for a single-lane bridge from opposite directiosn. The first to swerve away yields the bridge to the other; if neither player swerves, then the result is a potentially fatal collision.

The best thing for each driver is to stay straight while the other swerves; the worst is for a crash.

What will the drivers do?

We can consider a payoff matrix:

| | Swerve | Straight |
|---|---|---|
| Swerve | 0, 0 | 1, −1 |
| Straight | −1, 1 | −1000, −1000 |

Here, player 1's strategy is in the first row, and player 2's in the first column — so the first column corresponds to player 1 serving, and the second column to player 1 going straight; similarly the rows correspond to what player 2 does. (Sometimes the matrix will be laid out the other way, but this will always be specified.)

The two numbers in each cell correspond to the payoffs for each player (with player 1 first). For example, in the first row and second column, player 1 goes straight while player 2 swerves; so player 1 gets across the bridge and has a payoff of 1, and player 2 has to wait and has a payoff of $-1$.

First, we need to decide on our implicit assumptions. In this game, we'll assume the players don't cooperate. (There are situations where players cooperate, but this is outside the scope of this class — we will only consider *non-cooperative games*, where there is no mutually agreed upon strategy.)

We also assume that the players move *simultaneously* — not after seeing the other player make a move. (This is important because if player 2 first decides to swerve, then player 1 would decide to stay straight, while if player 2 first decides to stay straight then player 1 would serve. But we don't have this — they have to declare their moves at the same time. We could again consider games where this isn't the case, but this is again out of scope.)

We also assume that the players play *once* — this matters. For example, if they play a few times and we realize that player 2 is bullish and doesn't care, then we'd have to bias towards swerving. Again, repeated games are out of scope.

We also assume that the players have *all* the information — in particular, they know this matrix. So players 1 and 2 know for *themselves* how much they value each of the outcomes — player 1 knows that if they go straight and player 2 swerves, their payoff would be 1, and if they crash then the payoff is $-1000$. This is not always the case — you may know you want something without knowing your valuation of it. But furthermore, the *other* player knows your valuation as well. This is a fairly restrictive setting — these are called games of **perfect information**. Again, we could also consider games of imperfect information, which are also out of scope.

We also assume that all players are *hyper-rational* — they are rational, they know that the other player is rational, they know that the other player knows that they are rational, and so on. This does change things — if we didn't consider rational or hyper-rational players, the setting would be very different.

> **Question 13.2.** What will the players do?

We won't answer this yet. But a key idea is to use *randomness*. We've listed strategies that the players take, but they could also randomize their strategies — for example, they could toss a coin to decide whether to go straight or swerve.

### §13.1.1 An Easier Problem

Suppose we now have a different matrix:

| | Swerve | Straight |
|---|---|---|
| Swerve | 0, 0 | 1000, $-1000$ |
| Straight | $-1000$, 1000 | $-500$, $-500$ |

In this case, both players will go straight, because *regardless* of what the other player does, going straight is better (as 1000 is better than 0, and $-500$ is better than $-1000$). It looks like they should both select 'swerve' since that's better for both of them; but if we use the argument that they both do what's best for them given the other person's choice, then they both go straight.

In this case, the answer was more obvious than in the previous one.

### §13.1.2 Mixed Strategies

So far, we've talked about *pure* strategies, where you employ one strategy or the other. But we can also take *mixed* strategies.

Suppose player 1 swerves with probability $p$, and player 2 swerves with probability $q$. Note that these are independent — the two players can't communicate, so they won't tell each other what the outcomes of their coin tosses are.

The values of $p$ and $q$ would be known to each player — because of complete information — but the *outcomes* of the coin tosses are not known.

> **Student Question.** *How is this consistent with player* 1 *not knowing player* 2*'s strategy? For example, if* $q = 1$*, and player* 1 *knew that, then wouldn't they know player* 2*'s strategy?*
>
> **Answer.** We will get to this.

Then we can calculate the expected payoff of a player, using these values — since they're independent, we know the probability of being in each box.

## §13.2 Equilibrium

What we're trying to do is find a notion of equilibrium, where both players are happy to be where they are.

### §13.2.1 Dominant Strategy

A *best response* is to a strategy that has already been declared. For example, if I tell you that player 2 swerves, then player 1's best response is to go straight; meanwhile, if player 2 goes straight, then player 1's best strategy is to swerve in the original problem and to go straight in the altered one. We can read off the best response from the payoff matrix.

> **Definition 13.3.** A strategy is **dominant** if it is the best response to *any* strategy that the other player can take.

> **Example 13.4**
>
> In the modified problem, for both players going straight is a dominant strategy — no matter what player 2 does, player 1 should go straight. This is why we decided that they should both go straight.

> **Example 13.5**
>
> In the original problem, neither player has a dominant strategy — if player 1 says they're going straight, then player 2 is going to swerve, while if player 1 instead says they're going to swerve, then player 2 would switch and go straight. So no strategy works best in all scenarios. This is actually what makes this situation complicated to analyze.

### §13.2.2 Equilibrium

In the case where both parties have dominant strategies, they will both use their dominant strategies, and that'll be an equilibrium — neither player has an incentive to move. So for example, in the modified problem both players going straight is a *dominant strategy equilibrium* — both players have a dominant strategy, so they both follow it.

But what if we *don't* have a dominant strategy?

**Definition 13.6.** A strategy is a **equilibrium** if each strategy is a best response to the other.

For example, if player 2 swerves, then player 1's best response is to go straight. If player 1 goes straight, then player 2's best response is to swerve. So then player 1 always going straight and player 2 always swerving is an equilibrium — neither player has an incentive to deviate.

In pure strategies, we've seen what to do — we can look at every single entry and see 'if we're here, would either player want to move?' If both players want to sit where they are, then that's an equilibrium.

But what if we mix strategies? Then how would we talk about best responses?

Then we could optimize *over* $p$ and $q$. We can view every pure strategy as a mixed strategy with probabilities 0 and 1 — for example, if player 2 swerves (assigning $q = 1$ and $1 - q = 0$), then player 1's best response is to go straight (so $p = 0$ and $1 - p = 1$). Now we can view a certain $p$ as a best response to a certain $q$, and vice versa.

**Question 13.7.** What if player 1 picks a strategy with $p = 1/2$? What's the best-response $q$?

We can calculate that player 1's expected payoff is $p(999 - 1000q) + (1001q - 1000)$, and player 2's is symmetric.

Intuitively, the payoff for both to go straight is so low that a $1/2$ bias doesn't cut it. But if you went straight a very small fraction of the time, then I could consider going straight as well — because the chance we'd collide would be small.

## §13.3 Normal Form

Now we've seen a whole bunch of things, and we'll formalize and analyze them.

### §13.3.1 Normal Form Games

We define a normal form game as one where we have a set of players $\mathbb{P} = \{1, 2, \ldots, N\}$, and we use $\mathbb{P}_{-i}$ to denote $\mathbb{P} \setminus \{i\}$.

Each player has a set of actions $A_i$ (which may or may not be the same for all players).

The **action profile** is the list of all possible outcomes — $A = A_1 \times \cdots \times A_n$ is the set of all action profiles.

**Example 13.8**

In our example, the action profiles would be (straight, straight), (straight, swerve), and so on.

A **pure strategy** is one where player $i$ picks $a_i \in A_i$.

Each player also has a **utility (payoff) function**, defined as a function $u_i \colon A \to \mathbb{R}$ (from the set of all possible outcomes).

We can alo define a **mixed** strategy, where player $i$ picks $a_i$ according to some distribution $s_i$ on $A_i$. Then player $i$'s utility or payoff is defined by

$$\sum_{a \in A} u_i(a) \cdot \prod_{j=1}^{n} s_j(a_j).$$

(This simply calculates the utility of being in each cell, weighted by the probability that we end up in that cell — the probability that $a$ occurs is $\prod s_j(a_j)$, and $u_i(a)$ is the utility that player $i$ gets from that action profile.)

We have the same assumptions as before — the actions are simultaneous, and we have perfect information.

### §13.3.2 Best Responses

We use $s$ to refer to the set of all strategies, and $s_{-i}$ to refer to the set of all strategies employed by all players other than $i$.

We always define a best response for a *given* strategy — when we ask player 1's best response, we need to know *to what*.

**Definition 13.9.** We say that $s_i$ is player $i$'s best response to $s_{-i}$ (the set of strategies of $\mathbb{P}_{-i}$) if we can't replace $s_i$ by some other strategy $s_i'$ to get a better utility — in symbols,

$$u(s_i s_{-i}) \geq u(s_i' s_{-i})$$

for all strategies $s_i'$.

In other words, given $s_{-i}$ of $\mathbb{P}_{-i}$, player $i$ has no incentive to deviate from their strategy $s_i$.

Note that this definition includes mixed strategies. (In that case we are not given the actual outcomes — but we are given the strategies, meaning the probability distributions.)

**Student Question.** *Why are we using this framework, when we're not actually given anyone else's actions?*

**Answer.** When you played the game, you *would* make a calculated guess on what the other player would play, and think about what we'd do in that case. This is sort of how the optimization will work.

**Example 13.10**

In the original problem, if player 1 has $p = 1/2$ then player 2's expected payoff is

$$q\left(999 - 1000 \cdot \frac{1}{2}\right) + 1001 \cdot \frac{1}{2} - 1000 = 499q - \frac{999}{2}.$$

Player 2 should make this as big as possible; this means they should take $q = 1$.

**Exercise 13.11.** There is always a pure best response.

### §13.3.3 Dominant Strategies

**Definition 13.12.** A strategy $s_i$ is dominant if and only if it's the best response to every $s_{-i}$.

So player $i$ doesn't need to know anything about what the other players are doing — regardless of what the other players are doing, player $i$ has no incentive to deviate from $s_i$.

Of course, dominant strategies do not always exist.

**Exercise 13.13.** If there is a dominant strategy, then there is a pure dominant strategy.

This is slightly different from the previous statement — unlike a best response, a dominant strategy may not exist.

## §13.4 Nash Equilibria

**Definition 13.14.** A **Nash equilibrium** is one where no player benefits from unilateral deviation.

In other words, a Nash equilibrium is a collection of strategies $s_i$ (one for every player) where each strategy $s_i$ is a best response to $s_{-i}$.

**Example 13.15**

In our original problem, one person going straight and the other swerving is a Nash equilibrium — given that player 1 goes straight, player 2 won't switch from swerving, and given that player 2 swerves, player 1 won't switch from going straight.

This is the standard of equilibria for non-cooperative games — when we say equilibrium, we will mean a Nash equilibrium.

If the strategies are pure, then we call the Nash equilibrium pure.

**Theorem 13.16**

Every finite normal game has a Nash equilibrium.

Note that there exist finite normal games without a *pure* Nash equilibrium! But every finite normal game has a *mixed* one (possibly pure).

**Example 13.17**

In our modified problem, we had a *dominant strategy equilibrium*. This is also a Nash equilibrium.

Meanwhile, there are Nash equilibria that are not dominant strategy equilibria — in our original problem, both (swerve, straight) and (straight, swerve) are Nash equilibria.

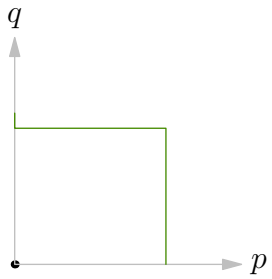
### §13.4.1 Mixed Nash Equilibriums

Suppose player 1 picks $(p, 1-p)$, and player 2 picks $(q, 1-q)$. Then player 1's payoff is

$$p(999 - 1000q) + (1001q - 1000).$$

We can see that if $q = 999/1000$ then the payoff is constant in $p$ — every choice gives the same outcome. If $q < 999/1000$, then we'd want to bump up $p$ as much as possible to maximize this, so we'd take $p = 1$. Meanwhile, if $q > 999/1000$ then the first term would be negative, so we'd set $p = 0$.

So then we can plot a *line of best responses* — the line of best responses for player 1 *given* player 2's strategy would be a green line:




At a Nash equilibrium, the two lines must intersect. The intuition we've seen earlier is that if you go straight fairly often, then it's always in my best interest to swerve — because the payoff from a collision is so negative — but if you have low probability of going straight, I may want to swerve. This formalizes that — we can see that there's three different Nash equilibria here, with one pure and one mixed.

**Example 13.18**

Consider a game of rock, paper, and scissors (where each player has a payoff of 1 if they win, $-1$ if they lose, and 0 if they tie). Then there is no pure Nash equilibrium. But there is exactly one mixed Nash equilibrium — each player choosing each with probability $1/3$.

At a mixed Nash equilibrium, we can see that I play an action so that the other player is *indifferent* to the actions that they play. Philosophically, that's because each player is trying to make sure the other player doesn't have incentive to deviate; and the way you do that is by making the other player *indifferent* to their actions. So here we'd look at player 2's possible actions, and the payoff of all their possible actions would be the same.

## §13.5 Zero-Sum Games

Nash proved that every finite normal game has a Nash equilibrium. (The proof uses the fixed point theorem — for certain classes of functions, they always have a fixed point. Most of the heavy machinery is in proving that theorem.)

We will look at a specific class of games, and use our LP-knowledge to prove that zero-sum games do in fact have Nash equilibria.

### §13.5.1 Setup

We consider the setting of two players, whre each player has a set of actions and $A = A_1 \times A_2$ is the set of action profiles, and player $i$'s utility payoff is defined by $u_i \colon A \to \mathbb{R}$ with $u_1 = -u_2$.

**Example 13.19**

Both of our swerving games are examples of this.

So then we can describe both payoffs in terms of a single matrix $U$.

We're trying to find a Nash equilibrium for a zero-sum game. Let's suppose that there's a mixed strategy for player 1, with probabilities $(x_1, x_2, \ldots)$ for all their actions, with $x_1 + x_2 + \cdots = 1$; and player 2 has the same. Then we want to find $x$ and $y$ such that $(x, y)$ is a Nash equilibrium — such that $x$ is a best response to $y$, and $y$ a best response to $x$.

**Question 13.20.** What is the payoff of player 1?

We have a matrix $U$, with player 1's strategies listed on top and player 2's on the left; and the payoff in the matrix is the payoff of player 1.

If player 1 uses probabilities $x_1$, $x_2$, ... of being in each cell, then we have a probability of $x_1$, $x_2$, ... of being in each column. So we'd multiply $U$ by the vector $x$ on the right — because then action 1 gets multiplied by $x_1$, action 2 by $x_2$, and so on.

Similarly, for player 2, we'd place $y$ on the left — because we're then multiplying $y_1$ with row 1, $y_2$ with row 2, and so on.

So this means the payoff for player 1 is $y^\intercal Ux$, and the payoff for player 2 is $-y^\intercal Ux$.

We can imagine player 1's thought proces — for any strategy that player 1 picks, if they told player 2 'Hey, I'm picking $x$,' then player 2 would pick $y$ to *minimize my payoff* (since maximizing their payoff and minimizing mine are the same thing in a zero-sum game).

So then player 1 is going to pick $x$ such that in this thought experiment, they'd have the highest payoff that they could — they'll pick $x$ such that the minimum possible payoff they'll get (over the choice of the other player) is as high as it could be. So player 1 picks $x = \arg\max_x \min_y y^\intercal Ux$ — we call this *maximin* (it's the max over the min).

Player 2 can run the same thought experiment — if they pick $y$ and tell player 1, then player 1 will pick an $x$ that maximizes their payoff $y^\intercal Ux$. So then player 2 is going to pick $y$ such that this maximum is minimized — they're going to pick $y = \arg\min_y \max_x y^\intercal Ux$. This is called *minimax*.

**Remark 13.21.** The notation arg min means the *argument* — when you're not tracking the value of the actual maximum, but rather the value of the variable that you put inside to get that maximum.

Let's look at the expression $\arg\max_x \min_y y^\intercal Ux$. In the 'min' expression, both $y$ and $x$ are known. As soon as we move out of the 'min,' $y$ is no longer known, and we're maximizing just as a function of $x$ — the expression $\min_y y^\intercal Ux$ is a function only of $x$ (you've already minimized over $y$).

Let's write $U$ as a series of rows

$$U = \begin{bmatrix} - & u_1 & - \\ - & u_2 & - \\ - & u_3 & - \\ \vdots & & \end{bmatrix}.$$

Then each row is the set of payoffs for player 1 assuming player 2 plays a corresponding strategy.

Now we're trying to compute $\arg\max_x \min_y y^\intercal Ux$. Let's try to look at the inner expression first. Look at any specific $x$. Then we can expand out $U^\intercal x$ — imagine we do this multiplication first. Then we have

$$\min_y y^\intercal Ux = \min_y y^\intercal \begin{bmatrix} u_1 x \\ u_2 x \\ \vdots \end{bmatrix}.$$

We're trying to minimize over $y$ this vector. Let's suppose, for example, that that vector is

$$Ux = \begin{bmatrix} 5 \\ -2 \\ 3 \\ 1 \\ 7 \end{bmatrix}.$$

Then to minimize

$$\begin{bmatrix} y_1 & y_2 & \cdots \end{bmatrix} \begin{bmatrix} 5 \\ -2 \\ 3 \\ 1 \\ 7 \end{bmatrix}.$$

Of course, we would do this by picking the entry that's actually minimal — we'd set $y_2 = 1$ and all other entries to 0. (We're essentially trying to induce a distribution on a set that minimizes our expected value, and the way to do that is to pick the minimum every time.)

So we have managed to drop out all the variables — we have

$$\min y^\intercal U x = \min u_i x.$$

Now we can write this as a linear program — we want to maximize the value of $v_1$ such that $u_i x \geq v_1$ for all $i$, with $\sum x_j = 1$ and $x \geq 0$. (If we have this, it's clear that we'd have the max-min; seeing the other direction is nontrivial.)

Here $v_1$ is an unconstrained variable in the primal; equality variables in the primal correspond to unconstrained variables in the dual, and vice versa.

And if we do this for player 2 — if we do the exact same exercise — then we'll get *exactly* the dual of this LP.

This means they have the same value. And by saying they're equal, we're saying that there's actually a solution to the problem we started out with; that solution is a Nash equilibrium.

**Theorem 13.22** (Minimax Theorem)

The minimax equals the maximin; and therefore zero-sum games have Nash equilibria.

This is interesting because it isn't always true — if we look at a general function, it won't necessarily be true. But it's true for this specific situation where we hit a matrix with one vector on one side, and the other vector on the other.

# §14 Random Walks

## §14.1 Intro

**Example 14.1**

Srini is on a trip at Las Vegas. He has \$20, and he will play until he is either broke or ends up with \$40. He wins a game with probability 1/2; when he wins he makes \$5, and when he loses he loses \$5. Will he make \$40 or not?

We could solve this problem right away, but we'll start off by looking at a few other approaches. One way to think about this is that if Srini wins is first four games, then he's done; so there's *at least* a $1/2^4 = 1/16$ chance that he wins. Meanwhile, there's also an outcome that guarantees he will lose — if he loses the first four games, then he loses. So there's also a probability of at least 1/16 that he doesn't make it big, which means the probability he makes it big is at most 15/16.

We could try to bound the probability from both sides — trying to list all the outcomes and seeing which direction they end (whether they hit 0 or 40 first), and calculating the probabilities and adding them up.

But as you can imagine, this is rather tedious — maybe you could find a pattern, but it'd be difficult.

Let's look at another way of modelling the problem. For each of these sequences, we know that if I need to win, then I need to get four more wins than losses (before I get four more losses than wins). So we can create a variable $p_{i,t}$ that denotes the probability of having $i$ more wins than losses at the end of game $t$. This is possible if at the end of the previous game I had $i-1$ more wins than losses, and then I won the current game. It's also possible if I had an additional win in the previous game, and I lost the current game. This means

$$p_{i,t} = \frac{1}{2}p_{i-1,t-1} + \frac{1}{2}p_{i+1,t-1}.$$

(We also have a few boundary conditions.) We could use this to compute all the probabilities in some systematic way — for example $p_{1,1} = p_{-1,1} = 1/2$ and all other probabilities $p_{i,1}$ are 0; then we could use the recurrence to calculate this for 2, then 3, and so on.

This is algorithmically at least easy to compute — if we gave a computer these values it'd spit out the value. Then we might say

$$\mathbb{P}[\text{Srini makes it big}] = \sum_{i=1}^{\infty} p_{4,t},$$

but this isn't quite true — our recurrence includes cases where we lose four games first.

### §14.1.1 Time

We've previously introduced the notion of time. Srini is an early sleeper, so he's also interested in another question — how *long* is he going to be playing? He hasn't set up a 'quit by game $x$' clause.

Let $q_t$ be the probability of finishing at the end of game $t$. Then $q_1 = q_2 = q_3 = 0$, $q_4 = 1/8$ (there's only two cases — we win or lose all four games). If we think about it carefully, $q_5 = 0$ — if for example we had three wins and then lost, then we've cancelled out one of the three wins and we have to add two more wins to finish. We could try to build a recurrence and take the expectation given the probability distribution; this would tell us on average, how long Srini woudl take. Of course, the hard part is 'what is $q_t$'.

Srini works in crypto; there are times when the protocols you design are so complicated that it's not possible to analyze them *well* nad *accurately*. You can be very loose and get some analysis, but it'll be weak; a tight analysis can be hard. So then we can simulate it — suppose we have a guess that our function behaves linearly, but we dont' knwo the constants. THen we can simulate it several times, fit it, find the constants and argue that they make sense, and this gives an empirical proof.

So let's say we simulate wins and losses with a coin toss.

In this simulation, in three of the four cases Srini made \$40. Let's say I ran this experiment several thousands of time; then I could calculate how long the game took, and whether I got \$40 or \$0, and just average everything. Srini recommends that we do it.

The other way to visualize this is by a meandering visual sum of hwo many wins and losses we have — we draw path where we take a step $(1,1)$ when we get a win, and $(1,-1)$ when we have a loss; then we're trying to hit the lines $y = \pm 4$. Then we have something randomly walking up and down, and we want to see whether we hit the top or bottom line first. When we hit either line we stop; we call this the line *absorbing*.

We could also imagine variants of this game — what if I start with \$25, and I end if I get \$35 (and the win probabiltiy is still 1/2). In this case, you're more likely to win — the closer I am to what I call winning, the more likely I should be to win. (My gain will be lesser if I do win, but I'll have a better chance of making that gain.)

In the 20-40 case, the chance that we win should be 1/2 — it's symmetric. (It's possible that the game goes forever, but this probability will go to 0 — it's a measure-zero set.)

Note that the game must end (with probability 1), and I must end with amount 20 or 40.

## §14.2 Markov Chains

### §14.2.1 Stochastic Process

**Definition 14.2.** A **stochastic process** is a collection of random variables indexed by time.

Mathematically, we can model this as one random variable for every time-instant — we can denote the process as $\{X_t \mid t \in \mathbb{N}\}$. In our picture, $X_t$ denotes the number of excess wins at the end of game $t$.

**Definition 14.3.** A **Markov process** is a *memoryless* stochastic process — future outcomes are based solely on the present state.

So we can essentially forget the entire history. If I've played 20 games, then determining the random variable for the 21st game only depends on the random variable for the 20th — because we know that with $1/2$ probability we get one more win, and with $1/2$ probability we get one less win. Recall that we had the recurrence

$$p_{i,t} = \frac{1}{2}p_{i-1,t-1} + \frac{1}{2}p_{i+1,t-1},$$

where no one brought in $t-2$ or $t-10$ (or for that matter, $t+20$).

We call this the **Markov property** — that future outcomes can only depend on the present state (or in other words, giving you the current state is as good as giving you the set of all the history).

### §14.2.2 Markov Chains

We can represent Markov processes graphically; we call this a Markov chain. Suppose that $\mathcal{X} = \{X_t \mid t \in \mathbb{N}\}$ is a Markov process, where all the $X_i$ come from a common domain $\mathcal{D}$. Then we can look at the states on a graph.

In principle, the probabilities can change (the win probability could be $1/2$ on the first game and $2/3$ on the next). So we really get a *collection* of Markov chains, drawing edges with the transition probabilities.

Usually we will look at *time-homogeneous* Markov processes — where $\mathbb{P}[X_{i+1} = v \mid X_i = u]$ is independent of the time $i$. Then we can draw *one* single directed graph $G_{\mathcal{X}}$, where we draw an edge $u \to v$ whenever $\mathbb{P}[X_{i+1} = v \mid X_i = u] > 0$, and we weight this edge with the probability.

Note that at each state, all the arrows going out of it must sum to 1.

If we have a graph (or a weighted graph), we can write down an adjacency matrix — where the $(i,j)$th entry is the weight of the $ij$-edge — the probability that we move from state $i$ to state $j$. For example, the row

$$\begin{bmatrix} 1/2 & 1/2 & 0 & 0 \\ 1/2 & 0 & 1/2 & 0 \\ 0 & 0 & 1/2 & 1/2 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

means that $a$ stays on $a$ with probabilty $1/2$ and goes to $b$ with probability $1/2$. Every row must sum to 1; the columns don't have to.

There is a way to take the adjacency matrix and compute paths between vertices. SUppose we had a true adjacency matrix, with 1's where we have an edge and 0's otherwise. Then if we take powers, if we take $k$ powers then teh $(i,j)$th entry will tell us the exact number of paths between vertices $i$ and $j$ of length $k$ .(This is just by matrix multiplication — we have to find $k-1$ vertices to go to in the middle, and the matrix multiplication will track those things.)

But what this is doing for a weighted graph is multiplying the weights of the edges on that path. In teh normal case, the weights are all 1, so the weight of a path is exactly 1 and we add them up. For general weights, we're taking teh product of weights on that path.

For weights which are probabilities, thsi makes sense — the weight of a path is the probability that we take the path (assuming independence).

So this means the adjacency matrix, taken to the $k$th power, gives the probability distributions — the first row gives the probability distriution of where I land up assumign I started at $a$. So it's weighting all the paths by the chance that you take that path — this tells you that at theend of 3 steps you're at $a$ with 3/8, $b$ with 1/4, $c$ with 1/4, and $d$ with 1/8.

Note that the row sums always remain 1 — because we have to remain somewhere in teh graph (we can't just disappear).

If we continue this process, then our numbers in each column start looking really alike — we can see that our matrix becomes something like

$$\begin{bmatrix} 4/9 & 2/9 & 2/9 & 1/9 \end{bmatrix}$$

in every single row. So something very cool has happened — after a while, no matter where we start, we end up in $A$ with probabiltiy 4/9, $B$ with 2/9, $C$ with 2/9, and $D$ with 1/9 (because all the rows look the same.)

> **Remark 14.4.** This won't happen for eveyr process — if we have a bipartite graph, then the thign won't converge, because we're on a different side after every odd or even step.

And the reason foro this, vaguely, si that fi I take this matrix and hit it again with the probability matrix, then I get back the same thing.

## §14.3 Random Walks

Let's say we have a Markov chain $\mathcal{X}$. Then the evolution of the process can be seen as a **random walk** on $G_{\mathcal{X}}$ — we start somewhere, the edge weights tell us how to select which edge to take, and we jus twalk randomly according to this probability distribution. (We take a particular step with the given probability.)

A **trajectory** is a specific walk — which we can think of as a sequence of state.s

> **Remark 14.5.** We can define a random walk on any type of graph — for an undirected graph we can add directed edges, for unweighted graphs we can assume all neighbors have the same weight.

If we look at what we did before, trying to list the outcomes of our game, we were listing *trajectories* — we were listing specific trajectories of the random walk induced by our Markov process.

We can start the walk at a specific vertex (which is usually what we think), but we can also start the walk at a *distribution* of vertices — we sample our first state according to some distribution, and then walk according to the probability distribution of the walk. (For a wakl starting at a specific vertex, our starting distribution has all the mass on one vertex.) And the probability distribution evolves over time.

More formally, a distribution over the vertices is a vector in $[0,1]^{|\mathcal{D}|}$.

The distribution evolves over time by multiplying it by the adjacency matrix — if we start at $B$, then we end up with a distribution of

$$\begin{bmatrix} 0 & 1 & 0 & 0 \end{bmatrix} \cdot \text{adjacency matrix}.$$

This is because if I start at a specific state and want to find the new distribution, then we just want a row (by definition).

But this generalizes — if we start with *any* distribution (writtne as a horizontal vector), then we can right-multiply by the adjacency matrix to get the probability distribution.

Now we could imagine taking the 80th power — this gives the probability distribution if we walk 80 steps. Then *any* probability distribution we hit it with to start with will produce almost the same result — no matter how I distribute myself at the start, I end up with teh same distribution after a certain number of steps (or at least very close to it).

**Definition 14.6.** A distribution is **stationary** if if we star tsomewhere and take a step, the distribution doesn't change at all.

So in other words, the distribution doesn't change under the action of our Markov chain — the vector times the walk matrix gives us back the same vector. (This is realted to eigenvalues and eigenspaces.)

So here $(4/9, 2/9, 2/9, 1/9)$ would be a stationary distribution. In general, a stationary distribution is a solution $(x_1, x_2, \ldots)$ such that if we right-multiply by the thing, we get back the same vector. (We also need the entries to sum to 1.)

In the example we saw, the Markov chain did move towards this stationary distribution. But this isn't always true — in a bipartite graph you may not even have a stationary distribution, and there may be graphs which have stationary distributions but you never get to them.

We can ask two questions:

- Does a stationary distribution exist ?
- Is it unique?

If a walk converges — it ends up at some distribution — then by definition, it must be a stationary distribution. So if a walk converges at all, it must converge to a stationary distribution. The question is, will it? ANd if there are multiple stationary distributions, which does it converge to?

**Theorem 14.7**

If $G_{\mathcal{X}}$ is *strongly connected* and *aperiodic* (the GCD of cycle lengths is 1), then a random walk on $G_{\mathcal{X}}$ converges to a unique stationary distribution.

Here cycle lengths are on the edges, ignoring weights. (Note that we only take edges which have positive weights — if there's an edge of weight 0, we wouldn't add it.)

Strongly connected is easy to check. Aperiodic may be harder —- but note htat self-loops definitely imply that the graph is aperiodic (this is actually a very common way to see something is aperiodic).

We also call *strongly connected* as *irreducible* — we can't reduce the graph into connected components.

First we can think about why these properties are important. Irreducible means that through the walk, I can spread myself across the graph (I can go to many different states).

What's important is that the random walk *will* convege, and it'll converge to a *unique* thing. So there is a stationary distribution (it exists), the stationary distribution is unique, and any random walk will converge to it.

**Remark 14.8.** All the things we've been talking about assume the Markov chain is fintie — there's a finite number of states. If there were instead infinitely many (i.e. the gambler doesn't stop at 40, but keeps going), then we need to add the condition that there should be a measure-nonzero chance that I keep visiting all the states. (After 10000 steps I shouldn't be stuck in half of the states.) THis is called *positive recurrence* — there i sa positive-by-meausre chance that you come back.

Note that there's a difference between convergence for *all* vs *some* initial distributions.

If the random walk converges for every starting state, then when we take the powers of the matrix, the rows must look the same. So this says that if we took the matrix corresponding to a strongly connected and aperiodic graph and took its powers, at some poitn all the rwos woudl start looking the same, and that would give the stationary distribution.

> **Question 14.9.** What if the conditions don't hold?

If $G$ is strongly connected, then $G$ has a *unique* stationary distribution. If $G$ is not strongly connected, then there may be vertices that you can't get to from one. Then you could imagine a stationary distribution on the different components. THe most trivial example is when we have two points which are both self-loops — everything is a stationary distribution.

So a stationary distribution may still exist (if you have multiple components), but you will have multiple of them.

So if you don't have strongly connected, it may not be unique; if you do then it will be unique.

If the graph is aperiodic, then a random walk on $G$ converges to a stationary distribution. If it's aperiodic but not strongly connected, every wakl will still converge, but differnet walks may converge to different distributions.

If $G$ were strongly connected but periodic —

In the undirected case, there is always a stationary distribution. But for a bipartite graph, the walks may not converge.

> **Exercise 14.10.** IF $G$ is connected, then it has a unique stationary distribution.

> **Exercise 14.11.** For an undirected graph, if $G$ is non-bipartite, then every walk on $G$ converges to a stationary distribution.

IF the graph is bipartite, then the walk may not converge.

> **Student Question.** Are there graphs with no stationary distributions?
>
> Yes but they are not easy to construct.

## §14.4 Back To Our Problem

We can model Srini's game as a Markov chain — the states are the amount of money he has (or the win differences).

0

Note that this graph is *not* strongly connected, since if we start at 0 or 40, we can't get anywhere else. But it is aperiodic, so the walk will converge to something.

If we take the 80th power, then it end up with a thing hwere all th middle stuff are 0, and the ends have different values for each row, between 0 and 1.

So after 80 steps you're basically never at any other state — eventually Srini will end up broke or make \$40 (he won't end up anywherein between). And we can see the answer to our problem is 1/2 —- if we start in the middle state, then we end upw ith either end state with probability 1/2.

To solve the problem, we can make a better recurrence. We can't solve tis problem using our theorem — we know it converges, but as an exercise we can try to figure out all the stationary distributions for this matrix; basically any distribution of the form $p$, $1-p$ ont eh two ends is stationary (and nothing else is stationary — all the middle entrie shave teo be 0). And we want to know what they converge to — this matrix doesn't have the structure where every row is the same, so it does depend on where we start. So the graph not being strongly connected i simpacting this — where we start the walk impacts hwere we end up.

NOw let $\alpha_i$ be the probability that you win starting with $5i$ dollars. So $\alpha_0 = 0$ and $\alpha_8 = 1$. Meanwhile, in between we have

$$\alpha_i = \frac{1}{2}\alpha_{i-1} + \frac{1}{2}\alpha_{i+1}$$

since the probability I win is half the probability I win after losing the first game, plus half the probability I win after winning the first game. This isi where the Markov property is important — I've moved one step t othe left in the chain, and now I'm asking the probabiltiy I win from here, and it doesn't matter how I got to that state, only where you are. So the probability can just be replaced with $\alpha_{i-1}$.

So this recursion is true because fo Markov. This is actually a pretty easy system of equations to solve; it gives $\alpha_i = i/8$. This matches our intuition that the more moeny you start with, the greater your chance of winning; and it matches the answer we got.

So this system of equations tells us that out of the infinitely many stationary distributions, this s the one we converge to if we start out with $5i$ dollras.

NOw suppose we take a different version of the game, where we change the Markov chain — instead of making our states absorbing, we say we can either stay there or move back (with a 50-50 chance).

Now this graph is strongly connected, so it has a stationary distribution. If you start somewhere and keep going left and right, you should be anywhere on teh chain with equal probabiltiy; and the powers of the matrix converge to the unifrom distribution on all vertices.

### §14.5 Mixing Time

> **Definition 14.12.** The mixing time is hte time it takes for a Markov chain to get 'close' to the stationary distribution.

Usually we go by the $L_1$ norm, so the numbers I get should be close to 4/9, 2/9, 2/9, 1/9.

Naturally, you can imagine that getting *closer* to the distribution will take more time.

ANother intuition for mixing time is from the name — what we want is for our walk to sort of mix over all the states. Then in the strongly connected aperiodic case, we want to sort of travel to all teh states enough that we can spread ourselves over the possibilities. So if the chain is longer, the mixing time will be longer (because it'll take longer for you to make your way across the thing).

## §15 November 1, 2022 — Random Walks II

### §15.1 Review of Last Lecture

Last time, we introduced the toy problem where Srini is at Las Vegas, and he starts with \$20 and plays until he reaches \$0 or \$40; each game gives him $\pm 5$.

We used this to discuss *Markov chains*, where future states only depend on the current state. We can describe a Markov chain using graphs, and in a *time-homogenous* Markov chain we can only use *one* (weighted and directed) graph, where the nodes indicate states and the edges denote transition probabilities. (Note that we only have edges of positive weight.)

Then we introduced the idea of a *random walk* — a random walk helps us understand hwo the Markov process that we started with evolves over time. At every step, we move from the current state to a next state based on the probability distribution of our neighbors; as we move in time, we're tracking a *distribution* of the set of states.

You could start either at a particular state, or at a distribution; then that distribution evolves over time by virtue of the adjacency matrix (or the walk matrix), whose entries are the transition probabilities.

Then we introduced the idea of a **stationary distribution**. For any Markov process, or any matrix, a *stationary distribution* is a vector that doesn't change when multiplied by the matrix. So in terms of the Markov process, taking one step doesn't change the distribution at all.

If the Markov process (or underlying graph) has some properties, then we saw that a random walk will converge to a stationary distribution. We saw two conditions of *strongly connected* (there is a directed path between every pair of vertices) and *aperiodic* (it's not that the walk would cycle in certain multiples — the gcd of all cycle-lengths is 1). Note that self-loops are a cheap way of getting aperiodicity; but it's very common.

For the infinite case, the same applies; but we also need the condition that at any point in the walk, the probability of getting from that state to every other state is nonzero.

> **Student Question.** *Does this condition automatically imply strongly connected?*
>
> **Answer.** Yes.

> **Student Question.** *Are there any Markov chains that don't have stationary distributions?*
>
> **Answer.** *Finite* Markov chains always have a stationary distribution. We've already seen in the undirected case this is true; in the directed case it is also true. But we may not converge to it.
>
> In the infinite case, this isn't true — if you have a chain where you don't come back to vertices infinitely often, then there isn't a stationary distribution.

In our toy problem, the graph wasn't strongly connected. But because the process was Markov, we could write down a simple recurrence and solve it. (It's just a linear system; it may look scary because it has a lot of variables, but you can often unwind it systematically.)

## §15.2 Expected Time to End

The question we didn't answer last time was *how long* Srini would be playing the game.

We'll try to solve this problem directly, because we've already noted that the chain we're discussing doesn't have strong connectedness. (Even for graphs that are strongly connected and aperiodic, where we know it'll convege to the stationary distribution, it would still *work* to write the recursion. But it would be easier to just find the distribution.)

> **Student Question.** *How would we find the stationary distribution without solving the recurrence?*
>
> **Answer.** By writing down the matrix equation.

Let $t_i$ be the expected time that Srini plays starting with $5i$. Then we have $t_0 = t_8 = 0$. Meanwhile, if we start with any amount, then either $i \mapsto i+1$ or $i \mapsto i-1$, either with probability $1/2$. This gives

$$t_i = \frac{1}{2}t_{i-1} + \frac{1}{2}t_{i+1} + 1.$$

(The $+1$ is because we need to count this step.)

Earlier without the +1 we got a linear answer. Here because the +1's compound, we get a quadratic answer; it turns out that

$$t_i = i(8 - i).$$

Unsurprisingly, this is maximized in the middle — if we're closer to one end, then it's more likely that the game will end soon. Specifically, when $i = 4$, the expected amount of time is 16.

> **Student Question.** *For a strongly connected graph, what do we mean by 'expected time to end' — the person will never stop?*
>
> **Answer.** There the question will be slightly different. In this case it's clear where the graph ends; in a strongly connected graph it may not be clear, but we will impose the condition to say that the walk ends when a specific condition is met.

## §15.3 Mixing Time

Our previous question was about reaching a specific state. But we can also ask — if we know the walk will converge to a stationary distribution, how long will that take? THis introduces the notion of *mixing time* — the amount of steps we need to take to get 'close' to the stationary distribution.

The idea is that after we take a certain number of steps, we say we're basically at the stationary distribution; once that has happened, we say that the walk has mixed.

The larger the chain gets (with all other parameters remaining the same), the longer it should take to mix — because the bare minimum we need is that the distribution has a chance to spread itself across the chain, and the longer the chain, the more steps it'll take to even get to the end. So if we have two linear chains of length 4 and 5, then taking 500 steps on the four-node chain gets us much closer to the stationary distribution than on the five-node chain.

How do we talk about mixing time? One method is *coupling*.

### §15.3.1 Coupling

In the recitation, we saw an example of card-shuffling, where we swapped two random cards. There, we were given an argument that as long as you see every card at least once, we say that the chain has mixed; roughly the argument is that once that card is touched, it goes to some random spot. So if every card is touched, then every card is in a random spot; hence we're at the stationary distribution.

That is an informal statement of what we'll talk about now, which is *coupling*. If we're given a random problem and asked for the mixing time, how would we figure out the condition of 'I want every card to be hit at least once?' This answers the question, and also shows that mixing isn't always easy to pin down — any Markov process you'll encounter in real life, it won't be hard to show that it converges, but seeing how tightly you can bound the time it takes to converge is much harder. (Sometimes we have bounds, but processes mix much better in process; and we have conjectures but don't know how to show them.)

The idea is in principle we could have two completely different Markov chains, and define independent random walks on them. But we could instead try to *relate* these random walks. So if we look at the walks independently, they'd be perfect random walks. But if we look at them together, they're not — we're making joint choices.

The important thing to note is that each walk independently would be a perfect random walk, but together they're correlated.

You can think about this in general — say we have Markov processes that converge to the uniform stationary distribution (for example, 10 states, with 1/10 on each). What this means is if we say we're at some step

20 after mixing. Then if you didn't tell me anything, the chance I'd be at any one of the states is 1/10. But if you told me who my previous neighbor was, then I'm no longer 1/10 — I'm limited by where I came from. So independently at every time-step the state looks uniformly random, but two consecutive states are correlated.

As a simple example, take a chain of length 3 (with $A \leftrightarrow B \leftrightarrow C$, and self-loops at the ends), and suppose the stationary distribution is $(1/3, 1/3, 1/3)$. Suppose we keep walking, and we have mixed.

Now suppose we take some arbitrary step afterwards, and ask what the probability we're at $A$ is — then it's 1/3.

But what if you now know that you had $A$ on the previous step? Then the chance you get $A$ is no longer 1/3. (This will be true even if you take a certain length, unless the length itself mixes.)

So even though things themselves may be uniform, jointly they may not be; and we will exploit that.

We will look at coupling in identical graphs — we take one graph $G$, and make two random walks. We say that they have *coupled* when we hit the same state in both walks.

We will not make these walks independently — we'll make them in tandem, so that each walk tells each other what it's doing. If you looked at one walk independently it'd look like it's making purely random choices, but in reality they're talking to each other. And if we can get them to collide then we'll say that they've coupled.

(For example, you could use the same random number generator to generate both.)

Suppose we start one walk, the walk we'ret rying to analyze how fast it mixes. Meanwhile we start a different walk at the stationary distribution. Once these two walks couple, we say that we have mixed. The idea is that because we're maintaining the distribution, the stationary distribution always has the stationary distribution (if we start a random walk with the stationary distribution, it'll always be in the stationary distribution).

Meanwhile, we want our walk to get to the stationary distribution. Once they've coupled, we can say we make the walks go the same way. Then our first walk didn't start as a stationary distribution, but now it's met the stationary distribution and become one; so we want to find the amount of time it takes to meet the stationary distribution, and then we're done.

(Note that we are looking at a specific walk; we're saying that the *walk-states* match, not the distributions.)

**Example 15.1**

Consider a bidirectional square graph with self-loops.

Suppose we start the left walk at a specific state $S$, and the walk on the right at a uniform state.

If the state we start at (randomly) starts at $S$ itself, then we're done. But we want to see how long it would take for that to happen in general.

One way of viewing this is assigning two bits to every state, where connected states differ by 0 or 1 bits.

Then we can describe the walk as — if we're at states $b_1 b_2$, then we sample a random position (either the first or second position) and set that index to a random bit 0 or 1. In two cases, the state doesn't change, accounting for the self-loop; in the two other cases, we move with probability 1/4. This gives us exactly the same chain.

But now it's easier to describe how we'll correlate the two walks. In the left chain we start with $(0, 0)$, and in the right chain we start somewhere random. Now the way we correlate is that we use the same position and bit choice for both walks.

The walk is always to look at your state, look at a bit, and assign that position to that bit.

Now if we look at the walks independently, they're each picking a uniformly random position and bit; so they're doing a perfectly normal random walk.

But if you pick a position, then you've picked the same position in both walks and assigned it to the same bit. So the states from that point onwards will match in that position from then on. (For example, if we start with 00 and 11, and then we pick the first position and set it to 1, then we have 10 and 11. After this point, the first position will always match in the two states — because we've made them the same right now, and we'll always make them the same from then on.)

So the walks will collide as soon as we pick both positions. This is a coupon collector problem — we have two coupons and pick one randomly, and we want to see how long it takes to get them both.

This can be extended to the card shuffling problem.

Now if we look at a different chain, we'll see that we mix when a particular graph hits 0, which is the graph we started out with.

## §15.4 Other Characteristic Times

So far, we've seen hitting time (how long it takes to hit a given state), the first return time (how long it takes for a walk starting at a particular state to get back to that state).

It turns out the stationary distribution tracks how long it takes you to get back. In the $(1/3, 1/3, 1/3)$ distribution, after we've mixed, we come back to each of $A$, $B$, and $C$ once every 3 steps, on average. This means $1/\pi_i$ is the first returntime.

Suppose we want to find $\tau_{b,d}$ in a certain example. Then we can write an equation $\tau_{b,d} = 1 + \tau_{b,a}$. Then we get the equation $\tau_{b,a} = 1 + \tau_{b,a}/2 + \tau_{b,b}/2$. Of course $\tau_{b,b} = 0$, and substituting back solves our system.

So if we started at $d$, on average it would take us 3 steps to get to $b$.

Similarly, we can find that the first return time of $d$ is 9; this is the inverse of $\pi_d$.

The hitting time for a chain is then the max of the time it takes to hit each vertex.

The **cover** time is how long it takes to hit all the states.

The hitting, cover, and mixing times are all related.

## §15.5 Markov Chain Monte Carlo algorithms

MCMC algorithms allow sampling *from distributions*.

> **Question 15.2.** Why is sampling a hard task?

One reason is if we don't know the distribution. Meanwhile, we might also know the distribution but it might be computationally hard — this was a big problem when the algorithm was initially introduced (it may not be now).

The idea was to sample distributions using *fast* machines — because a lot of the time, distributions have a normalization constant. We know how probabilities of different states are related to one another, but we don't know the sum over the entire state space, which we need in order to characterize the distribution. And even if we do have it as a scary integral which we can approximate, at that point they didn't have that — so it wasn't feasible to actually do this. So they needed cleverer methods to do that task.

The goal is to design a Markov chain with the desired stationary distribution; and we want a MCMC algorithm that is rapidly mixing.

## §15.6 Metropolis–Hasting Algorithm

Suppose we have an arbitrary distribution that we don't know how to sample from, but we do know how the probabilities of two states are related — and that's all. (We just need to be able to compute a function *proportional* to the probability distribution at that point.)

The cool thing i sthat even though you can't sample, you will pick another distribution that you do know how to sample, and somehow use it to approximate the distribution at hand — you have a target distribution you don't know how to sample, and a proposal sample which you do know how to sample (which may be unrelated), and somehow you're going to use the proposal distribution to sample the original.

What we're going to do is define a Markov chain that is *reversible* (more restrictive than strongly connected and aperiodic, so even more well-behaved so that we can easily show that it converges to the target distribution).

What we do is — we start at any state, and then we use our proposal distribution to sample a next state. If we knew the actual target distribution, then we would sample using the target; but we don't, so we sample using the proposal instead.

And then we use the target distribution to evaluate how good our guess is — we sample $x'$ according to $g(x' \mid x)$ ($g$ is the proposal, $\nu$ the target). Then we compare $\nu(x')$ and $\nu(x)$. We compute

$$p_{\text{acc}}(x', x) = \min\left\{1, \frac{\nu(x')g(x \mid x')}{\nu(x)g(x' \mid x)}\right\},$$

and then we move to $x'$ with probability $p_{\text{acc}}$ and stay at $x$ with $1 - p_{\text{acc}}$.

So all we need is the ratio of $\nu(x)$ to $\nu(x')$ — we don't need to integrate over the whole state space.

You can show that this Markov chain will actually converge to the distribution $\nu$. If we want to actually figure out the distribution, then you can run the Markov chain for a long time, assume we have some bound on mixing, and then we know we're at the stationary distribution $\nu$; so you can use a walk of very long length to calculate the actual values of $\nu$.

For example, if we have a very long walk *abbcaddaccdabc*..., then we could then say that we're not sure about the first five things so we discard them, but from then on we assume that the walk has mixed, so we've gotten to the stationary distribution. Then we can count how many times we hit $a$, $b$, $c$, and $d$; that frequency tells us the exactt stationary distriution. So not only can we sample from it, but we can actually understand it. (Of course there are issues with correlation; here our samples are not independent, so maybe you can get from $d$ to $c$ only in three steps. There are other sampling methods to do that, but they are more expensive.)

(A symmetric MH is when $g(x \mid x') = g(x' \mid x)$, so that the terms cancel.)

> **Student Question.** Why do we need $g$ here?
>
> We don't have $\nu$ normalized. For example, we might have $(1, 1, 1, \ldots)$ but not $(1/n, 1/n, \ldots)$ — this carries as much information (it tells us the stationary distribution), but we can't actually build the Markov chain. $g$ is important because it's an actual probability distribution that we can sample.

Here $p_{\text{acc}}$ is '$p$-accept' — MH is basically you have a current state, guess a sample for the next state, and then evaluate. $x'$ is your trial, and you evaluate it and decide to accept it with probability $p_{\text{acc}}$ and reject otherwise. If the two $g$'s are the same, then this means whichever of $\nu(x')$ and $\nu(x)$ is larger, you go to that node.

**Student Question.** How would we calculate $g(x' \mid x)$?

We're assuming that $g$ is sample-able — that we can draw samples from it. This means we have an algorithm that produces values according to that distribution — a sampler for the uniform distribution would be a random number generator. So when we say $g$ is sampler, we have some algorithm that produces distributions according to $g(x \mid x')$.

The goal is to sample according to $\nu$ (we don't need to produce something that's actually a random walk).

We are not given $g$; we will have to determine it. (Picking $g$ is part of the algorithm design.)

In the min expression, we can ignore the $g$'s; then we're just looking at $\nu(x')$ and $\nu(x)$ and comparing them (the only reason we have the min is so that if we're higher than 1 then we actually get a probability back).

(The right way to think of the given big expression is $(\nu(x')/g(x' \mid x))/\ldots$. The term is introduced to accomodate more proposal distributions, which may not be symmetric but give you better things)

No matter what $g$ you use, it'll converge; but some stuff do it better. (This is why Hastings introduced the correction factor — to account for general distributions which may not be symmetric.)

### §15.6.1 Choosing g

A common choice of $g$ is to make a random walk using some other distribution — for example, move to a random neighbor. You get to pick the proposal distribution, so you can draw any arrows you want; normally we will try to draw arrows to every other neighbor and perform teh standard undirected walk.

The intuitive idea behind this is that we explore the state space via perturbations. Basically we're at some point, and we look in our neighborhood to see if there's a better point; if we're there then I move there. Now the issue is how big of a step I should take.

In the continuous case, you generally walk along a normal distribution. For example say we have a bimodal distribution. Now if our perturbation is relaly small and we start in one peak, then there's no way we'll get out of it — every time we go left, the algorithm pushes us right. So we only find one of the peaks. But if we take too wide a step, then we don't recognize the peaks at all. So the parameter that you use for MCMC has to be closely monitored; many times, the way you evaluate the propsal distribution is to run many walks, and se if you're getting a simliar distribution every time. If there's something you're missing, it'll show up in some trial; and you'll see that you need to tune $g$ better to capture that.

THe point is not that we're converging to the wrong distribution — we can prove if you ran it longer, it would find both peaks. But it would have to be run *really really long* — because the chance you'd make the transition is really small.

The other key point is that it's often hard to bound mixing time — many times you won't have a concrete number of how long you should run it to have mixed.

**Example 15.3**

Suppose we want to sample a random graph coloring.

This comes up a lot in graph theory (and in statistical physics — a lot of MH applications are there).

We have an undirected graph, and we want to assign a color to every vertex such that adjacent vertices don't have the same color.

As a related problem, we want to sample a *random valid graph coloring.*

Given a coloring, we can tell whether it's a good coloring (we check adjacent vertices).

You could go over the space of all colorings and fliter out good and bad ones; this is very hard. THe state space is very large, an dthis problem of counting exactly the valid colorings on a graph is very large.

MH means that even though we don't know the exact number of colorings, we can sample a random one.

Imagine you had all the valid colorigns listed out, but you don't know how many of them there are. (Exactly as how we have $1, 1, 1$ but we don't know the normalization constant is 3.)

**Algorithm 15.4 —** We start with any valid coloring (this is not hard — just try coloring, and correct mistakes every time you make them, somewhat systematically).

Now pick a random vertex and pick a random color. See if you can switch that vertex to that color. If you can, do this; if you can't, then we reject teh trial and go again.

This is parallel because we have some distribution to pick a coloring starting from an original — pick a random vertex, random color, assign it. And then we have acceptance probability 'check if it's a valid coloring'. If it's a valid coloring, we'd have the value 1, so we move to it — $\nu$ just evaluates to 1 if valid, 0 otherwise.

So using $g$ we sample, using $\nu$ we evaluate, if it's good then we move to it.

The magic is that if I've already told you MH converges to the normalized $\nu$, well $\nu$ weights every valid coloring 1 equally. So if you look at the space of all valid colorings, the distribution is uniform — I've weighted proportionally every valid coloring the same. THis means in a distribution sense, it's the uniform distribution on that many colorings. And MH will converge to that distribution.

**Student Question.** What if MH meets after ten times the state space?

Say a chain has a million states. You don't usually need a million steps to mix. This is a weird thing you should think about, but you don't — because in a distribution sense, it is moving out faster. In fact, mixing time is related to the *diameter* of the graph — if there exist ways to get to the end of the grpah, that's enough. So even though a graph may have really long *longest* paths, if your graph is very connected then the diameter (longest shortest path) is small.

**Student Question.** How do we decide when it's mixed?

There is more theory that goes into this (and sometimes you just do experimental stuff).

## §15.7 Reversibility

A **reversible** Markov chain has the property that if $\pi$ is a stationary distribution, then $\pi_u W_{uv} = \pi_v W_{vu}$ for all $u$ and $v$.

$\pi_u$ is the probability you're at $u$ in the stationary distribution, and $W_{uv}$ is the probability that you go from $u$ to $v$. So this says the chance you'r eat $u$ and then move to $v$ is the same as the chance you're at $v$ and then move back to $u$ (int eh stationary) — we're moving across $uv$ in both directions.

**Exercise 15.5.** Random walks on weighted graphs are reversible, but not all Markov chains are reversible.

In fact, if $\pi$ achieves this, then it has to be stationary (it implies that $\pi W = W$ — we're not just saying that $\pi W = W$, we're saying that every term in the sum is equal.)

So if we went all the way to mixing, then we could walk the chain in either direction and it would look the same — it'd look liek it's coming from teh same Markov chain.

The reason to mention this is because MH you can show is actually reversible. (IT's a simple exercise —- you define $W_{xx'}$ as $g(x' \mid x)p_{\text{acc}}$, and you can show that $\nu_x W_{xx'} = \dots$ — in other words, $\nu$ witnesses the stationarity of the walk using $W$, which means $\nu$ is the stationary distribution of $W$. This is the proof of why MH works — not only have we constructed a Markov chain, we've constructed a special one which is reversible and has stationary distribution $\nu$. THis is why we have thos econstants.) Ignoring the min, ew get $\nu(x)W_{xx'} = \nu(x)g(x' \mid x)p_{\text{acc}}(x', x) = \nu(x')g(x \mid x') = \nu(x')W_{x'x}$.

# §16 November 3, 2022 — Intractibility

## §16.1 Introduction

Today we'll talk about problems that are difficult — in this case, polynomials for which we don't have a polynomial time solution. What we'll learn today and next lecture is how computer scientists have come to develop a framework for thinking about these difficult problems. In this class, we only know what we can prove — anything we suspect or think might be true doesn't count. And what we'll see is that for these intractable problems — these problems that haven't yet succumbed to a polynomial-time solution — it's still possible to create a framework for proving things about them.

Today we'll set up a lot of infrastructure to deal with that framework. We'll talk to different classes of problems — we're used to optimization problems, but it'll be beneficial to talk about search problems, and even more simply, decision problems. We're going to learn to classify things as polynomial, or as another category NP (*non-deterministic polynomial time*) — P problems can be solved in polynomial time, and NP is a superset of P along with some other grouping of problems we'll discuss.

We'll learn about the tool of *reductions*. We'll see a group of problems that are of special interest — the NP-complete problems — and we'll see how to use reductions to prove a problem is NP-complete. And then we'll prove that a certain problem is NP-complete.

The topic starts as being a bit counterintuitive, but as we see how it's applied, it'll make a lot of sense.

## §16.2 What Makes a Problem Easy?

In many situations, there's pairs of problems where one version is easy to solve, but for a slight variant, we don't have a polynomial time solution — even though people have tried hard.

**Example 16.1**

In 6.006, we learned about the shortest path problem — to find a shortest simple path from $s$ to $t$ given a weighted graph.

But if we change the problem from the *shortest* path to the *longest* path, no one's been able to find a polynomial time algorithm.

**Example 16.2**

We've learned about minimum spanning trees — to find a spanning tree of minimial weight — and we've seen Kruskal's and Prim's algorithms.

Meanwhile, the Travelling Salesman problem is similar — given the same input, instead of finding a spanning tree, we want to find a spanning *cycle* of minimal weight. This doesn't look much harder, but no polynomial time algorithms are known, and people have really tried.

**Example 16.3**

We've learned about linear programming — if the vector $x$ is *real*, we can find polynomial-time solutions to maximize $c \cdot x$ subject to constraints $Ax \leq b$.

But if we require $x$ to be an *integer* vector instead, then we do not know a polynomial-time solution.

**Example 16.4**

We've seen how to do 2d-matchings — given a bipartite graph connecting students to internships, we can see whether there's a perfect matching (a way to assign each student to an internship).

But if we add one more grouping, to have a tri-partite graph — where we now add edges between an internship and a company if a company wants to work on that project — then there's no polynomial-time solution to find whether we can find a perfect matching (into triangles).

It feels like we're missing something, or there's something hard here. What we're missing is some algorithmic intuition about what change to a problem can make it easy, and what change makes it hard. For example, changing minimum spanning trees to maximum spanning trees is trivial, but changing it to minimum spanning *cycles* is unsolved.

Until we find a polynomial time solution, we don't know anything — we don't have proofs that say there *is* no polynomial time solution. And if we can't prove anything, then we can't say anything about these problems; that's not okay.

What we're going to learn is what we *can* prove, and how that might be useful.

It's a major open problem in computer science whether these intractable problems actually have a polynomial time solution that we haven't yet found. But what *we* can do is look to see whether we can prove, early on, that a problem is *similar* to these other problems for which we don't know a polynomial-time solution — to prove that it's a hard problem. And then we may as well not put in the effort to find a polynomial solution, because even though we can't prove that it doesn't exist, it may not.

## §16.3 Types of Problems

First let's formalize our problems into three types of problems — optimization, search, and decision.

**Example 16.5**

Consider the minimum spanning tree problem. As an *optimization* problem, given $G = (V, E, w)$, we want to find a spanning tree of minimum weight.

A *search* version of the same problem would be: given a graph $G = (V, E, w)$ as well as a budget $k$, find a spanning tree of weight *at most* $k$ (if one exists).

**Question 16.6.** Which of these is harder?

If we have an algorithm that can solve the optimization problem (as we do), then we could use it to solve the search problem — find the minimum spanning tree, and then check if its weight is at most $k$. So a solution to the optimization problem is directly a solution to the search.

Meanwhile, if we could solve the search problem, then we could binary search on $k$ to find the *minimum* — we could keep lowering $k$ until there's no solution, and then raising it a bit until we get a solution, and that must be the minimum.

**Student Question.** *What if there's a faster solution to the search problem? We know that the search problem isn't harder than the optimization problem, but we haven't proven that it's not easier?*

**Answer.** We know that the search problem isn't *harder*, because if we have a solution to the optimization problem, then it's directly a solution to the search problem. But that's really all we can say.

We also have a *decision* version:

**Example 16.7**

Given a graph $G = (V, E, w)$ and a budget $k$, in the *decision* version of MST we simply want to decide whether there *exists* a spanning tree of weight at most $k$.

Similarly to before, if we had a solution to the optimization or the search version, we could use it to solve the decision version. So the decision version is no harder than the search version, which is no harder than the optimization version.

Why discuss this? We care about showing that algorithms are hard, and the whole field focuses on showing that *decision* problems are hard — because if the decision version is hard, and it's the least hard of the set, then the search version is *at least* as hard as the decision version, and the optimization version is at least as hard as the search version.

**Example 16.8**

Some more examples of decision problems:

- Is there a path of length at most $k$ from $s$ to $t$?
- Is there a simple path of length *at least* $k$ from $s$ to $t$?
- Is there a spanning tree of weight at most $k$?
- Is there a spanning cycle of weight at most $k$?
- LP: Is there a real vector $x$ with $cx \geq k$ and $Ax \leq b$?
- IP: Is there such an integer vector $x$?

## §16.4 Polynomial Time

**Definition 16.9.** A decision problem $\Pi$ is solvable in **polynomial time** if there exists a polynomial time algorithm $A$ such that for every instance $x$ of this problem, $x$ is a YES input for $\Pi$ if and only if $A$ outputs YES.

**Example 16.10**

For the MST problem, an instance $x$ would be a graph, with vertices, edges, and weights, and the budget $k$.

So a polynomial-time problem is one for which there is a polynomial-time algorithm that says YES if and only if it's handed a YES instance.

**Definition 16.11.** We define P as the set of problems solvable in polynomial time.

So if $\Pi$ is solvable in polynomial time, then we write $\Pi \in \mathrm{P}$.

The problems we've seen so far — shortest paths, MST, linear programming, and 2d-matching — are all members of P.

## §16.5 Non-Deterministic Polynomial Time

**Question 16.12.** What can we say about problems like longest path or TSP or integer programming?

So far, all we've said is that we don't have a polynomial time algorithm — there might be one that we haven't found yet, or there might not be.

Often when we're given a problem and we play with it for a while, we find a piece of math that's true about it, and we use the math to build and then prove the theorem. So we need to be able to say something mathematical about these algorithms.

It turns out there is something to be said, that's extraordinarily useful although it may seem trivial. This formalism is beautiful because there's so little you could say without it, but with it you can say a lot, and prove quite a bit.

**Definition 16.13.** The set **non-deterministic polynomial time** (NP) captures problems for which we can *verify* a YES instance in polynomial time, with a polynomially short certificate.

**Example 16.14**

If you give me a TSP problem, I can't in polynomial time find out whether there is a simple cycle of weight at most $k$. But if you hand me a certificate, which in this case is a valid Hamiltonian cycle of weight at most $k$, then I can check it in polynomial time.

This is a tiny statement — I can't solve the problem at all, but I can *check* an answer in polynomial time.

**Remark 16.15.** The time is with respect to the input length — in this case, the specification of the graph and the value of $k$. So we need to be polynomial in that size. When we say a polynomially short certificate, we mean that the cycle you give me should also be of size polynomial in the input. Of course it is, because it has at most all the edges and all the vertices in it.

That's a tiny statement, but we'll see soon that it's extraordinarily useful.

First we need to formalize this further.

**Definition 16.16.** We have $\Pi \in$ NP if there exists a polynomial time verification algorithm $V_\Pi$ and a constant $c$ such that for all instances $x$ of $\Pi$, $\Pi(x)$ is a YES instance if and only if there exists a certificate $y$ such that $|y| \leq |x|^c$ and $V_\Pi(x, y) =$ YES.

**Remark 16.17.** NP does *not* mean 'not polynomial.' The reason for the name is that you could guess and check the answer in polynomial time.

In particular, not only are the hard problems on our list verifiable in constant time, so are the easy problems. So it's not that P and NP are on opposite sides.

**Proposition 16.18**

$P \subseteq NP$.

*Proof.* If $\Pi \in P$, we can take our verification algorithm $V_\Pi$ to simply be the same as the algorithm that solves the problem — that says YES if it's a YES instance, and no otherwise. This then satisfies the condition for verification. □

So there's a set P and a set NP it lives in, and both live inside a bigger set of *all* decision problems.

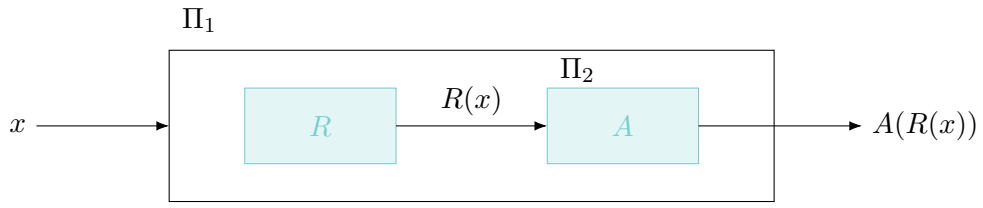
If it turns out that there's no problems in the space between P and NP, then they'd be the same set. This is a major open problem in computer science — whether P = NP.

## §16.6 Reductions

We learned about reductions when we reduced the bipartite matching problem to a max flow problem. To reduce means essentially to convert one problem to another — then we can borrow the solution to the second problem to help us solve the first.

Here we can do that, but we can also do something else — we can use the fact that we *don't* have a polynomial time solution to one problem to show that we don't have one to the other problem.

Suppose we have an outer problem $\Pi_1$ (such as TSP), with instance $x$; its output is then YES or NO. Then we could have a second problem $\Pi_2$ (such as integer programming). A *reduction* takes an instance $x$ of $\Pi_1$ and converts it into an instance $R(x)$ of $\Pi_2$. And then if we have an algorithm for the second problem $\Pi_2$, we can use it, and it'll give us an answer to the outer problem.




We want our reduction $R(x)$ to take polynomial time. If we have a polynomial-time reduction, and a polynomial-time solution to the inner problem, then we automatically have a polynomial-time algorithm for the *outer* problem.

This is useful for two things:

- If we have a polynomial-time algorithm for $\Pi_2$, we can use it to obtain one for $\Pi_1$.
- If we already believe that $\Pi_1$ is a hard problem, then if we can reduce it in polynomial time to the inner problem, $\Pi_2$ can't be easy — if it were easy (i.e. has a polynomial-time algorithm), then this would violate our assumption that $\Pi_1$ is hard (i.e. there's no polynomial time solution).

In this unit, we'll use reduction for the second case.

> **Definition 16.19.** A **polynomial-time reduction** from $\Pi_1$ to $\Pi_2$ is a polynomial-time algorithm $R$ such that if $x$ is an input to $\Pi_1$, then $R(x)$ is an input to $\Pi_2$, and $x$ is a YES instance to the outer problem if and only if $R(x)$ is a YES instance to the inner problem.

So a reduction converts one problem to the other, in such a way that the outputs also match.

## §16.7 NP-Completeness

We'll now define two groupings of problems — NP-hard problems and NP-complete problems. This is a way of representing groups of difficulties, in some sense.

**Definition 16.20.** A decision problem $\Pi$ is **NP-hard** if for all problems $\Pi' \in \text{NP}$, $\Pi' \leq_P \Pi$.

**Notation 16.21.** The notation $\Pi' \leq_P \Pi$ means that we have a polynomial-time reduction from $\Pi'$ to $\Pi$, so $\Pi$ is at least as hard as $\Pi'$.

So a problem is NP-hard if it's at least as hard as *every single problem* in NP.

**Definition 16.22.** A problem $\Pi$ is **NP-complete** if it is NP-hard and in NP.

This gives a way of classifying the 'most difficult' problems.

## §16.8 Cook's Theorem

WHat we'd really love to do is show that we have one NP-complete problem — because if we can do that, then we could reduce other problems to it. Then if our problem is at least as hard as that problem, and that problem is at least as hard as all problems in NP, then our problem is also at least as hard as all problems in NP.

Suppose we have some problem $\Pi_{215}$ that we've shown is NP-complete. Then this implies that for all problems $\Pi \in \text{NP}$, we have $\Pi \leq_P \Pi_{215}$.

But if we have a new problem $\Pi_{10}$, and if we can use reduction to show that $\Pi_{215} \leq_P \Pi_{10}$, then we know that

$$\Pi \leq_P \Pi_{215} \leq_P \Pi_{10}.$$

So just by *one* comparison to a NP-complete problem, we can establish that $\Pi_{10}$ is harder than *everything* in NP. (We also need to show that $\Pi_{10}$ is in NP, but that's usually easy.)

This means it's really important to find the first problem that we can prove is NP-complete — then we can use this formalism to prove that many are.

But how do we get the first one? What we want is one problem that *all* NP-complete problems reduce to.

This was done by Cook (1971) — he showed that a problem called *circuit-SAT* is NP-complete. Once he had this first problem, then others could use that as leverage to demonstrate that more problems are NP-complete — Karp (1972) showed that 21 more problems were NP-complete, and by now the list of NP-complete problems has grown into the thousands.

First Cook needed to choose the problem.

**Question 16.23.** Imagine a circuit made up of boolean logic gates — with two inputs into each AND and OR gate and one into each NOT gate, and infinite branching on the outputs.

AND

We have a set of inputs $x_1$, $x_2$, $x_3$, …, $x_n$, and some combination of gates with as many interconnections as we want, and a single output. Both the input and output are binary variables.

We assume that there's no feedback, so the graph is a DAG.

The decision problem is: given a circuit $\mathcal{C}(x_1, \ldots, x_n)$, is there an input for which the output of $\mathcal{C}$ is 1?

**Theorem 16.24** (Cook, 1971)
Circuit-SAT is NP-complete.

**Remark 16.25.** The proof is in the lecture notes, but Prof. Tidor thinks that it's not going to give us much insight.

To show this, we'd first have to show that circuit-SAT is in NP — that we can verify a YES instance in polynomial time with a polynomially short certificate. One such certificate would be a set of 0's and 1's on all edges — then we could verify in polynomial time that the logic is properly propagated through all gates and all edges.

And then we want to show that circuit-SAT is at least as hard as every problem in NP — this means we want to reduce every problem in NP to circuit-SAT.

The trick is that all we know about those problems is that because they're in NP, they have a certificate that can be verified in polynomial time. And what Cook did was to say that that's all we need — the reduction is essentially going to *be* the verification algorithm, and we know that the verification algorithm runs in polynomial time.

And then we can take that verification algorithm, and write it in code. And code runs on a computer, that essentially has a binary circuit inside. You can demonstrate a formalism that allows you to convert every verification algorithm into a piece of code that can be run on a circuit, and you can design a circuit that would run that code — with just AND, OR, and NOT gates. The hard work is connecting all the dots to prove that this works, but the insight was picking the first problem to be circuit-SAT.

# §17 November 8, 2022 — Intractability II: Reductions

Reductions are the primary tool we use to show problems are NP–complete. Today, we'll review some of the notions from last class about NP–completeness, and then we'll look at a series of reductions that are very helpful. As a guiding framework, what Prof. Tidor really likes about this formalism is that it lets us build provable arguments around problems that are too difficult currently — it allows us to prove things that we can prove, without being complicated by questions we don't know the answers to (whether there really are polynomial-time algorithms).

## §17.1 NP–Completeness

**Definition 17.1.** The set NP consists of all decision problems (problems with yes/no answers) whose 'yes' instances have polynomially short certificates that can be used to verify they are a 'yes' instance in polynomial time.

**Remark 17.2.** Here *polynomial* means polynomial in the input size.

**Example 17.3**

NP includes problems such as whether there's a 2D matching in a graph (which we know how to solve in polynomial time) as well as whether there's a 3D matching (which we don't); whether there's a path of length at most $k$ as well as whether there's a path of length at least $k$.

## §17.2 Reductions

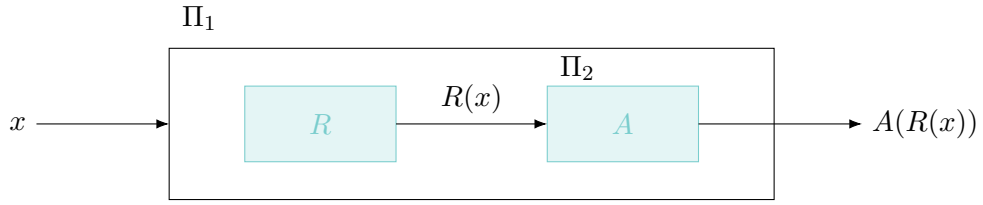
**Definition 17.4.** We say that $\Pi_1 \leq_P \Pi_2$ if there exists a polynomial-time reduction algorithm $R$ that does two things:

- It converts the input of $\Pi_1$ to an input of $\Pi_2$ — if $x$ is an input to $\Pi_1$, then $R(x)$ should be an input to $\Pi_2$.
- $\Pi_1(x) =$ yes if and only if $\Pi_2(x) =$ yes.

**Student Question.** *For the second condition, would it also suffice to check that $\Pi_1(x) =$ yes $\implies \Pi_2(x) =$ yes and $\Pi_1(x) =$ no $\implies \Pi_2(x) =$ no?*

**Answer.** Yes, because the second statement is the contrapositive of the other direction.

We can use reductions to show that a problem is NP-hard, and if it's also *in* NP, then it's NP-complete.




Last time, we said that the definition of NP-complete is essentially the 'hardest' problems in NP:

**Definition 17.5.** A problem $\Pi$ is NP-complete if and only if:

- $\Pi \in$ NP;
- It is at least as hard as all problems in NP — more formally, for all $\Pi' \in$ NP, we have $\Pi' \leq_P \Pi$.

The second condition alone defines a problem as **NP-hard** (a NP-hard problem is at least as hard as all of NP, but doesn't itself have to be in NP).

Last time we stated Cook's theorem, which gave us our first NP-complete problem:

**Theorem 17.6** (Cook's Theorem)
Circuit–SAT is NP-complete.

There's a much larger list of NP-complete problems, and today we will expand our list.

**Student Question.** *Are there problems that are NP-hard but not NP-complete?*

**Answer.** If there are problems that are not verifiable in polynomial time, those would be candidates. The halting problem may be an example.

## §17.3 SAT

**Question 17.7.** Given a Boolean formula, is it satisfiable (is there a set of 0 and 1 values to the variables that we can assign so that the formula itself evaluates to 1)?

**Example 17.8**

An example formula might be

$$\phi = (x_1 \vee \overline{x_2}) \wedge x_3 \wedge (\overline{x_3} \vee x_1 \vee x_2).$$

This is an AND of ORS; for the formula to evaluate to 1, we need each of the components to evaluate to 1. This means we must set $x_3 = 1$. Setting $x_1 = 1$ then satisfies both the first and last clauses, so we're done (and $\overline{x_2}$ could be 0 or 1). So this formula is satisfiable, and we were easily able to show that it's satisfiable.

**Definition 17.9.** A **formula** consists of $n$ boolean variables $x_1, \ldots, x_n$, with $m$ boolean connectives (which can be AND, OR, NOT, IMPLIES, IFF) and parentheses (which are important to get the groupings right).

**Student Question.** *What's the input size?*

**Answer.** The size of the entire formula — the $n$ boolean variables (with multiple instances of some of them), the $m$ boolean connectives, and the parentheses (because they're part of your input). It's really the amount of storage you need to describe the input.

**Theorem 17.10**

SAT is NP-complete.

To prove this, we need to first show that it's in NP, and then that it's at least as hard as all the problems in NP.

**Claim —** SAT is in NP.

(This is the easy part, but it's part of the definitions so we have to check it; so it's good practice to do it first.)

*Proof.* We can simply take a list of the satisfying variables assigned (as our certificate). This is polynomially short — it's equal in size to the number of variables. We can then evaluate the formula with that assignment by running through it in polynomial time. So this satisfies the requirements for being in NP. □

The next thing we want to show, of course, is that SAT is NP-hard — that for all $\Pi \in$ NP, we have $\Pi \leq_P$ SAT.

That would be really hard, so we instead use a trick — we already have one problem that we know is NP-hard, namely CIRCUIT–SAT. So instased of showing this, we'll instead show that

$$\text{CIRCUIT-SAT} \leq_P \text{SAT}.$$

Then by transitivity we already know that $\Pi \leq_P$ CIRCUIT-SAT $\leq_P$ SAT.

**Student Question.** *What's the difference between SAT and CIRCUIT–SAT*

**Answer.** CIRCUIT-SAT takes a circuit rather than a formula; in particular we can have unbounded fan-out.

We'll prove this by doing a reduction from CIRCUIT–SAT to SAT. This means we need to take a circuit and convert it into a formula.

The question just asked suggests that maybe, we can just take the circuit and re-express it as a formula. If we think about it, that infinite fanout ruins everything — it makes it possible that the formula can become exponential in size — we'll get a very nested formula.

So instead of converting into a formula that represents *exactly* the circuit, we will do something a bit different.

We start by adding new variables to the system (which are not inputs to the graph so are not labelled in the circuit) — one at each wire. This is polynomial in the input size.

First, the formula checks whether the output is 1. Meanwhile, we have a clause at each gate checking whether it performs the correct logic — for example, $x_8 \iff (x_5 \vee x_7)$ checks that $x_8$ is really the OR of $x_5$ and $x_7$ (as it is supposed to be); similarly $x_7 \iff (x_5 \wedge x_6)$ checks whether the AND gate is evaluated correctly. And so on.

So we're checking whether the output of the circuit is 1, and each gate performed its correct logic.

> **Student Question.** *Are the wires considered part of the input size?*
>
> **Answer.** Yes.

> **Student Question.** *Can you repeat why we can't just write out the equation for the circuit directly?*
>
> **Answer.** If you have enough fan-out (meaning that when a gate has an output, it connects to a lot of new inputs) and enough layers, you'll end up with an exponentially large formula when you're done (with a lot of parentheses).

This satisfies impedence matching — the formula is of the form it needs to be in. The harder part to state is that a 'yes' instance of the circuit produces a 'yes' instance of the formula, and vice versa. We'll take this in two steps.

> **Claim —** If the circuit is satisfiable, then the formula is satisfiable.

*Proof.* A valid assignment of the original circuit will produce a certificate — a valid value in each wire causing an output of 1. So if we put this into the formula, then the formula will produce 1 as well (because the output is 1, and each clause represents that a single gate has performed the correct logic). □

> **Claim —** If the formula is satisfiable, then the circuit is satisfiable.

*Proof.* If we apply our initial variables into the circuit, then the circuit itself will produce the matching remaining variables — because at each gate we'll be producing the correct logical output. Then by the first clause of the formula, we're ensured that the output will be 1. □

Combining these, we have that SAT is NP-hard, and therefore NP-complete.

> **Remark 17.11.** Here it wasn't hard to guess what to reduce from. Soon we will know enough NP-complete problems that part of problem solving will be coming up with a good problem to reduce from.

## §17.4 3-SAT

> **Question 17.12.** Given a formula $\phi$ in CNF with 3 literals per clause, is $\phi$ satisfiable?

A CNF is an AND of clauses, a *clause* is an OR of literals, and a literal is a variable and its complement.

**Example 17.13**

$(x_1 \vee \overline{x_2}) \wedge x_3 \wedge (\overline{x_3} \vee x_1 \vee x_2)$ is not in 3-CNF, because the first two clauses don't have three literals. Meanwhile, $(x_1 \vee \overline{x_2} \vee x_3) \wedge (x_2 \vee x_3 \vee x_4)$ is a valid input to 3-SAT.

**Theorem 17.14**

3-SAT is NP-complete.

First, 3-SAT is in NP because given values of all the $x_i$, we can check in polynomial time whether the polynomial is satisfied, and that's a polynomially short certificate.

It's easy to do a reduction of the form we're *not* looking for — it's easy to see that 3-SAT $\leq_P$ SAT (because SAT encompasses any boolean formula), but we need a reduction the other way around to show that 3-SAT is NP-hard — we need to reduce a *general* formula to the *specific* form.

We will not do this, in the interest of time.

So now we have three NP-complete problems

$$\text{cSAT} \leq_P \text{SAT} \leq_P \text{3-SAT}.$$

**Remark 17.15.** The statement 3-SAT $\leq_P$ SAT is straightforward because any input of 3-SAT is automatically an input for SAT, so our reduction can simply be the identity. But what about the other way around? If we have a general formula, it's more complicated to put it into 3-SAT.

The second point is that SAT genuinely is a harder problem than 3-SAT (it's all of 3-SAT and more) — or rather, it's at least as hard. If Prof. Tidor doesn't tell us what Karp did, we might be justified in thinking that SAT is really hard, and 3-SAT could be easier — so we could solve 3-SAT by doing a polynomial-time reduction to a much harder problem, but there might be another algorithm for 3-SAT that doesn't go through SAT and is easy. (There isn't, but any time we have a special case, we have to recognize that this construction doesn't require that the special case be as hard as the general case — only when we've set up the reduction in the right order can we prove that the special case isn't polynomially distinctly easier.)

## §17.5 Vertex Cover

So far, everything we've done is boolean.

**Question 17.16.** Can we use these boolean problems to reduce to other problems, such as graph problems?

**Question 17.17.** Given a graph and an integer $k$, does there exist a subset $S \subseteq V$ such that $|S| \leq k$ and every edge $e \in E$ is incident to at least one vertex in $S$?

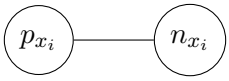
So we're given a graph with vertices and edges and a budget, and we need to find a set of vertices that covers all the edges.

Clearly VC is in NP — if you give me a graph and $k$, and the set of vertices that make up the vertex cover, then I can check in polynomial time that you gave me at most $k$ vertices and every edge is covered by at least one of these vertices (and the certificate is polynomially short because it's just a list of vertices).

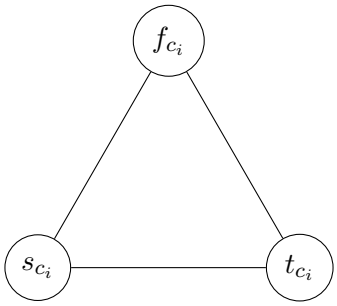
We're now going to show that 3-SAT $\leq_P$ VC, which will show that it's NP-hard. When we create a reduction, the input to the reduction is a formula in CNF with $n$ variables and $m$ clauses, where each clause has 3 literals. The goal of the reduction is to somehow create a graph $G = (V, E)$ and $k$ that encodes this formula — such that a satisfiable equation always produces a graph for which a vertex cover of size at most $k$ exists.

*Proof.* The technique that we'll use is called **gadget construction**. What we'll do is — for each variable we'll make a subgraph and for each clause we'll make a subgraph, and we'll connect them together in a graph to represent the formula that we have.

Each variable $x_i$ will be represented by a graph with two vertices (a positive and a negative) connected by a single edge.




Meanwhile, each clause will be represented as a triangle — with one vertex for the first literal, one for the second, and one for the third.




Note that in the variable-shaped gadget, choosing one vertex is necessary and sufficient to form a vertex cover for our subgraph. Likewise, for the clause gadget, choosing *two* vertices is necessary and sufficient.

**Example 17.18**

Suppose we have a formula

$$\phi = (x_1 \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee \overline{x_3}).$$

This formula has three variables, so we'll have three variable gadgets. It also has three clauses, each in parentheses; so we also have three clause gadgets below. (These numbers in general do not need to be equal.)

We're not done yet — we are going to add more edges. But with the graph that's here, the minimal number of vertices we'd need to put in a cover is 3 for the vertex edges, and 6 for the clauses, so 9 in total. So our vertex cover has to have at least $n + 2m$ (in general) — one per vertex and two per clause. We are going to set $k = n + 2m$.

**Student Question.** Are we assuming that the literals in each clause are distinct?

Prof. Tidor doesn't think that's necessary; even if they are repeated, everything will be fine. (We still set up three vertices for that clause.)

We haven't used anything about our formula yet — just the number of formulas and variables. So of course we have to do more work.

The thing we're going to do is: the first clause has as its first literal $x_1$ (not $\overline{x_1}$, but $x_1$). So we connect its first vertex $f_{c_1}$ to the positive vertex $p_{x_1}$. Then ew have $x_2$, so we connect $s_{c_1}$ to $p_{x_2}$; and similarly we connect $t_{c_1}$ to $p_{x_3}$.

We can then keep on going — in the second clause the first literal is $\overline{x_1}$, so we draw an edge from $f_{c_2}$ to $n_{x_1}$; the second literal is $\overline{x_2}$, so we draw an edge to $n_{x_2}$; and the third is $x_3$, so we draw an edge to $p_{x_3}$. And so on.

We know how we're going to cover all the white edges — we need to pick one vertex from each dumbbell and two from each triangle, and that covers all the white edges. What happens to the yellow edges? Let's look at the first clause.

We're going to pick two vertices, and that covers two of their yellow edges. But for the third one, that won't be covered by anything in the triangle. So it'll only be covered if we choose the positive end of one of its variable gadgets. That'll play itself throughout — to cover all the variable edges, we can cover two from the clauses, but we need to cover the third from having a variable.

So we choose $p_{x_1}$ if $x_1 = 1$ — then this creates a mapping between teh formula and graph, in such a way that we only get a vertex cover with this limited amount of vertices if the formula is satisfied. □

We've seen already that VC $\in$ NP. Now we need to show the other part, that 3-SAT $\leq_{\mathsf{P}}$ VC. We've described a way of constructing a graph based on the formula that's an input to 3-SAT, so we know that any valid formula creates a valid graph (by construction). Now we have to just show the iff parts: that if there exists a vertex cover of our graph of size $k = 2m + n$, then that implies $\phi$ is satisfiable, and vice versa.

Let's check the first direction first.

Suppose we color in the vertices $p_{x_1}$, $n_{x_2}$, and $n_{x_3}$. If there exists a vertex cover of size $k$, we know that it has to use exactly one vertex from each variable gadget and two from each clause gadget. The matching we're going to do is that let's pretend these shaded vertices are in the vertex cover. Then we set $x_1 = 1$, $x_2 = 0$, and $x_3 = 0$.

We want to show this assignment satisfies every clause.

Because we have a valid cover, at least one of the edges to every clause is going to be satisfied by the variable row above — for example, in the first thing $p_1$ satisfies the first literal in teh clause.

So because of the way the grpah is constructed, the fact that one of these yellow edges is covered by the variables means that one of these clauses is satisfied. We can go through and say the same for the remaining clauses. So our construction requires that at least one literal in every clause must be satisfied.

**Student Question.** Do you know all solutions have one from the top and two from the bottom?

For the white edges, having one from the top and two from the bottom is necessary and sufficient. When we put in teh larger graph, no other vertices can affect that edge. We don't know when we add all the yellow edges that this alone will be sufficient — it's possible there's a formula with no vertex cover.

**Student Question.** What vertices get selected in the triangle gadgest?

The vertices of the triangle gadgets whose yellow edges we have not covered with our variable gadget vertices.

**Remark 17.19.** Our example barely got satisfied. It could have been the case that some edge got covered from both ends, but that's fine. We would still need both our vertices at the bottom to be covering all the white edges.

Now let's show the other direction. If the formula is satisfiable, we want to show there exists a vertex cover.

If you hand me a satisfying assignment, then we color the top row according to that assignment, and we follow those in and color the *other two* vertices and add those to the cover. This works because at least one of these literals is a 1, so that covers at least 1 of our yellow edges leading into the clause, and we can cover the other two in this way. So a satisfiable formula has a satisfyign assignment, and we can use that to show that the graph has a vertex cover of size 9.

**Remark 17.20.** So far we've reduced

$$\text{cSAT} \leq_P \text{SAT} \leq_P \text{3SAT} \leq_P \text{VC}.$$

But we also know that $\text{VC} \leq_P \text{cSAT}$, since we've seen it's at least as hard as everything in NP. So equality must hold — all these problems are equally difficult. Likewise, if we had a polynomial time algorithm for any one of these problems, we could construct a polynomial-time algorithm for all of them by following the string of reductions.

**Remark 17.21.** Engineers need to solve problems that are NP-complete. Some ways of doing this are approximation algorithms, intelligent exponential search, average case analysis (algoirthms that work well most of the time), or working with special input cases.

# §18 November 10, 2022 — Approximation Algorithms I

Today we'll discuss the vertex cover, set cover, and partition problems.

## §18.1 Why Approximation Algorithms?

In the last few lectures, we saw there's a set of problems — NP-complete or NP-hard problems — that we don't have polynomial-time solutions for. In that case, what do we do?

One thing we can do is realize that we don't have $n \to \infty$ — our problem is a certain size — and we can still apply enough computing power to solve our problems.

But another approach is to try to get *close* to the right answer, guaranteed to run in polynomial time.

In general, we want to solve hard (NP-hard) problems, we want to use fast algorithms (in polynomial time), and we want to obtain exact solutions (correctly). But so far, we can only do two of these. In most of this class, we focus on (2) and (3) — we take problems that are solvable in polynomial time, and we obtain fast and correct solutions. This is why we focus so much on proving runtime bounds and correctness.

Today and in the next lecture, we'll talk about instead doing (1) and (2) — we'll solve NP-hard problems in polynomial time, allowing ourselves to just get *close* to the right answer instead of the *exact* right answer (where we can provably bound the error).

We generally require that the solution we get be a valid solution — if we're looking for a minimum spanning tree, then we should get a spanning tree (which may or may not be minimum). Similarly, for vertex cover, we may not get the minimum vertex cover, but we should certainly get *a* vertex cover.

In the previous two lectures, when we looked at intractability, we focused on the *decision* version of problems — the argument was that search and optimization were even harder. In approximation algorithms, we really do care to look at the optimization versions of the problems.

## §18.2 Some Formalism

**Definition 18.1.** Given an optimization problem of size $n$, we let $c^*$ be the 'cost' of the optimal solution, and $c$ be the 'cost' of our approximate solution. We then define the **ratio bound**

$$\rho(n) = \max\left\{\frac{c}{c^*}, \frac{c^*}{c}\right\}.$$

Note that by *cost* we don't mean runtime, we mean the thing we're optimizing — for example, in the longest paths problem, we mean how long the path we get is.

If we're working on a minimization problem, then we'll have $c^* \leq c$, so the left-hand ratio will be the greater one; if we're working on a maximization problem, it's the other way around. So the ratio bound is always some number at least 1.

This will be how we describe approximation algorithms. There's something similar that we call an *approximation scheme*, and we keep these things separate. An approximation scheme takes an additional parameter $\varepsilon$, the size of the acceptable error:

**Definition 18.2.** A **approximation scheme** takes as input some $\varepsilon > 0$ and provides a $(1+\varepsilon)$-approximation algorithm.

**Definition 18.3.** A **polynomial-time approximation scheme** gives an algorithm that's polynomial in $n$, but not necessarily in $1/\varepsilon$ (for example, $n^{2/\varepsilon}$).

**Definition 18.4.** A **fully polynomial-time approximation scheme** provides an algorithm that is polynomial in both $n$ and $1/\varepsilon$ (for example, $n/\varepsilon^2$).

We'll see some examples today; the logic for how to get the algorithms and how to get the bounds is especially neat, and somewhat counterintuitive.

## §18.3 Vertex Cover

Last lecture, we looked at the decision version of Vertex Cover; today we'll look at the optimization version.

**Question 18.5** (Vertex Cover)**.** The optimization version of vertex cover is as follows:

- Input — a graph $(V, E)$.
- Output — a set of vertices $S \subseteq V$ such that for all edges $e \in E$, we have $S \cap e \neq \emptyset$. In other words, for every edge in the graph, at least one of its endpoints is in the vertex cover.
- Objective — minimize $|S|$.

So a vertex cover is a set of vertices such that every edge has at least one endpoint in the cover.

One idea is to be greedy — to grab vertices with the highest degree. Another is for each edge, to choose a random endpoint (either uniformly, or based on the degrees of the endpoints).

All of these are good ideas; but how do we evaluate them? First, we want it to be easy to prove what our algorithm does — what its error bound is, and that it's polynomial in runtime. There may not be any logic telling us whether to start with the edges or to start with the vertices; we'll actually do *both* today.

**Algorithm 18.6** (2-approximation for VC) — Start by picking any edge $(u, v) \in E$, and initialize $S$ to be empty. Then:

- Add *both* $u$ and $v$ to the cover.
- Remove all edges from $E$ that are incident to $u$ or $v$ (since we have already covered them, and we don't have to worry about them anymore) — this removes the selected edge, and hopefully more.
- Repeat until $E$ is empty.

This wouldn't be our first guess — it might seem somewhat wasteful to add *both* $u$ and $v$. But let's notice a few things.

First, the runtime of this algorithm is polynomial — if we use an adjacency list to store the graph, then this takes $O(V + E)$.

The algorithm is also *nondeterministic* — it depends on the order in which we pick edges (if we pick edges in a different order, we could get a different vertex cover).

Also, the output $S$ is a valid vertex cover — we know this because an edge is only removed from the edge-set when it's been covered by at least one vertex that we put in the vertex cover, and we keep on going until the edge set is empty.

**Claim 18.7** — We have $|S_{\text{APX}}| \leq 2\,|S_{\text{OPT}}|$.

First, where does the 2 come from? Every time we select an edge, we need to choose *at least* one of its vertices, and we choose 2.

This proof is interesting because we don't know very much about either the approximate or optimal solution, but we can still argue about them anyways.

*Proof.* Let $A$ be the set of edges picked by the algorithm. Then we know that the edges in $A$ are 'free edges' — they never share an endpoint with each other. (If there was an endpoint where two edges met, then when one of them was selected and the endpoint put into the cover, the other edge was removed.)

Now we know that

$$|S_{\text{APX}}| = 2\,|A|\,.$$

(Both our ends are going to be in the cover — there are no shared vertices.)

On the other hand, all the edges in $A$ are also in $E$, so the optimum has to cover all of them. And since they don't share any endpoints, the optimum has to have at least one vertex from every edge in $A$. This means

$$|S_{\text{OPT}}| \geq |A|\,,$$

and putting these together, we get

$$|S_{\text{APPROX}}| \leq 2\,|S_{\text{OPT}}|\,.$$

□

**Remark 18.8.** This is an easy proof — the logic is very straightforward (although not necessarily easy to come up with!). So even though we did something that seems a bit wasteful, it led to a very nice proof.

We might imagine that picking one endpoint might be smarter. But if we picked a random endpoint, it wouldn't be — for example, take a star (where the optimal cover has 1 vertex). Then if we're unlucky and for every edge we pick the outer point, we'd pick $n - 1$ vertices.

**Remark 18.9.** We want our logic to cover the worst case, not the average case; expectation can be a different story.

We might also try to be greedier in the edge-selection, by picking edges whose endpoints have high degree. It turns out this doesn't improve the worst-case bounds (and leads to a messier proof with the same answer).

Note that we were able to show we had a 2-approximation without ever determining $|S_{\text{OPT}}|$ — this is very often the case (since in hard problems, we don't have the exact answer).

One thing that *does* help is post-processing — if we have extra vertices in the end, then we can remove them. This can be useful in practical situations.

## §18.4 Set Cover

Set cover is another NP-hard problem. In words, we have a set $X$ of elements (as black dots), and $m$ subsets of these elements (as colored circles).

**Definition 18.10.** A **set cover** is a collection of these sets so that their union includes all the elements in the larger set $X$.

Our goal is to minimize the number of sets we need to take.

**Example 18.11**

Suppose our elements are variants of a virus, and our sets represent things we can put into a vaccine in order to protect against that variant. Then we'd want the smallest number of spike proteins that are representative of the entire population of viruses.

In our example, we have one element *only* in $S_2$, so we need $S_2$ in order to catch all the elements; similarly we have an element unique to $S_3$, and one unique to $S_4$. So a cover will need to have $S_2$, $S_3$, and $S_4$, and that turns out to be sufficient.

More formally:

**Question 18.12.** Set cover is as follows:

- Input — a set $X$ of $n$ points and $m$ subsets $S_i \subseteq X$ such that $\bigcup_{i=1}^{m} S_i = S_1 \cup S_2 \cup \cdots \cup S_m = X$.
- Output — a cover $C \subseteq \{1, \ldots, m\}$ such that $\bigcup_{i \in C} S_i = X$ and $|C|$ is minimized.

In this case, we could try to do this greedily (always picking a set that covers the most elements). We could also try the logic that we did in this example, but that might not work in general (and it might get very messy).

**Algorithm 18.13** (Greedy Algorithm) — Repeat until all elements are covered:

- Choose a new set $S_i$ containing the *maximum* number of uncovered elements.
- Add $i$ to the cover.
- Mark all elements in $S_i$ as covered.

Finally, return our cover.

So we start with an empty cover and add subsets to it until every single element is covered; in particular, we certainly return a valid set cover.

To analyze runtime, every time through the loop, we remove at least one set from the set of possibilities; and we also remove at least one element. It's unclear whether we run out of sets or elements first, but we'll run through the loop $\min(m, n)$ times. Meanwhile, to find the cost of each iteration, we may have to run through all the sets and all the elements, giving $O(mn)$. This means our algorithm takes $O(mn \cdot \min(m, n))$ time. In particular, this is polynomial.

**Question 18.14.** How good an approximation is this?

Again, we don't know that much about our solution or the optimal solution; but somehow we need to find some mathematical solution between them (similarly to the previous problem).

**Claim 18.15 —** Greedy Set Cover is a $(\lfloor \log n \rfloor + 1)$-approximation.

The idea behind the proof is that at each iteration, a set is going to be chosen that will cover a large fraction of the number of elements that are left. We just have to see what 'large fraction' means and how we're going to prove it.

In the previous proof, we simply defined $A$ to be the set of edges selected by the approximation algorithm — we asserted that there's some part of the approximation result that we can pull out and start to analyze. Here we'll do the opposite — we'll assert something about the *optimum* and argue about it.

Let $C_{\text{OPT}}$ be the optimal cover, with $|C_{\text{OPT}}| = t$.

Then there has to be some set covering at least $n/t$ elements — we have $n$ elements in our set, and somehow we're able to cover them with $t$ sets (which may or may not overlap). So in some sense, the average set has to cover $n/t$ elements. (If some of them are doing less than their share of work, others are going to have to do more than their share; otherwise you wouldn't be able to cover all $n$ elements.)

This gets at the idea that there must be a set covering some significant fraction — and that significant fraction is given in relation to the size of the optimal cover.

*Proof.* Let $|C_{\text{OPT}}| = t$. Then at each iteration $i$, let $X_i$ be the remaining uncovered elements. Then we know $X_i$ can be covered by $t$ sets — because the optimum can cover all the elements, and at future steps there's even fewer elements, so of course the same number of sets can cover them.

This means there has to exist at least one set that covers at least $|X_i|/t$ elements (our substantial fraction) — because otherwise we couldn't cover the $X_i$ elements with $t$ sets. (If we have 50 elements and the biggest set only covers 2 elements, we couldn't cover them with 5 sets.)

Our algorithm necessarily picks such a set — it picks the biggest one, so it's going to pick a set that has *at least* $|X_i|/t$.

That's the key mathematical relationship. The rest is just arithmetic: at every step, we have

$$|X_{i+1}| \leq \left(1 - \frac{1}{t}\right) |X_i| .$$

Doing this repeatedly, we then have

$$|X_i| \leq \left(1 - \frac{1}{t}\right)^i |X| .$$

Now let's look at the step $i = t \log n$. Then

$$\left(1 - \frac{1}{t}\right)^{t \log n} \cdot n \leq e^{-\ln n} \cdot n = 1.$$

So when we get to step $\lfloor t \log n \rfloor$, the number of vertices left is at most 1 — which means the number of iterations required is at most $\lfloor t \log n \rfloor + 1$.

Meanwhile the optimal solution is at most $t$. This gives us

$$|C_{\text{APX}}| \leq (\lfloor \log n \rfloor + 1)\,|C_{\text{OPT}}|\,,$$

as desired. □

## §18.5 Partition

Partition is another NP-hard problem.

**Question 18.16.** The problem Partition:

- GIVEN — a sorted list of $n$ positive numbers $s_1 \geq s_2 \geq \cdots \geq s_m$.
- FIND — a partition of the indices $[n]$ into two sets $A$ and $B$ such that $[n] = A \cup B$, and $\max\{\sum_{i\in A} s_i, \sum_{i\in B} s_i\}$ is minimized.

So we're essentially trying to seek the most balanced partition — to make our two pieces have sums as close as possible.

**Student Question.** *Why is this problem in NP?*

**Answer.** We've only defined NP on decision problems. This is not a decision problem. That's why we've only said it's NP-hard, and not that it's NP-complete.

Here we'll actually get an approximation *scheme* — a parameter that we can tune, so that with more work we can get a tighter bound.

First, consider this polynomial-time approximation scheme:

**Algorithm 18.17 —** Define $m$ to be minimal so that $\varepsilon \geq 1/(m+1)$ (in other words, $m = \lceil 1/\varepsilon \rceil - 1$). Our algorithm has two phases:

- Find the optimal partition $(A', B')$ for the first $m$ elements (which are the biggest elements). Since $m$ doesn't depend on $n$, this can be done in constant amount of time (via brute force); if we want a tighter bound, we make $m$ bigger.
- Then we handle the rest of the elements — first initialize $A = A'$ and $B = B'$. For the remaining elements $i = m+1$ to $n$:
  - If we currently have $w(A) \leq w(B)$, then we add $i$ to $A$.
  - Otherwise, we add it to $B$.

So we first correctly sort the first $m$ elements, and then we take the remaining elements one at a time and add them to the smaller side.

This does return a valid partition.

**Question 18.18.** How good of an approximation is this?

**Claim 18.19 —** This is a $(1+\varepsilon)$-approximation.

*Proof.* Without loss of generality let $w(A) \geq w(B)$ at the end, and let $2\ell = \sum_{i=1}^{n} s_i$ — so then $w(A) \geq \ell$. If we also make $A$ bigger in the optimal solution, then it's also true that $w_{\text{OPT}}(A) \geq \ell$.

Then we can define our approximation ratio as

$$\frac{w(A)}{\ell} \geq \frac{w(A)}{w_{\text{OPT}}(A)}.$$

The proof focuses on the *last* index added to $A$.

**Case 1** ($k$ is added in the first phase). This implies $A$ hasn't changed at all since the first phase, so $A = A'$ (all future elements were added to $B$). This implies that we have the optimal partition for all $n$ numbers — because $A'$ was optimal for the first $m$, and the remaining $m - n$ were added in a way that maximally reduced the remaining imbalance.

**Case 2** ($k$ is added in the second phase). Then the algorithm tells us that at the time $k$ was added, we had $w(A) - s_k \leq w(B)$ (where $w(A) - s_k$ is the weight of $A$ before $k$ was added). But then this must also be true after termination, since $w(A)$ must remain the same and $w(B)$ can only grow larger.

So then we have

$$w(A) - s_k \leq w(B) = 2\ell - w(A),$$

which means that

$$w(A) \leq \ell + \frac{s_k}{2}.$$

But we have $s_1, \ldots, s_m \geq s_k$, and so

$$2\ell \geq (m+1)s_k$$

because $k > m$. This means we have

$$\frac{w(A)}{\ell} \leq 1 + \frac{s_k}{2\ell} \leq 1 + \frac{1}{m+1} \leq 1 + \varepsilon. \qquad \square$$

This proof is quite neat because we choose a particular element and focus on what the situation was when it was added, and how it's changed since that time.

## §18.6 Greedy Vertex Cover

In the remaining time, we'll look at a greedy vertex cover algorithm.

**Algorithm 18.20** (Greedy Vertex Cover) — Repeat until the edge set is empty:

- Pick the vertex $v \in V$ with maximal degree, and add it to the cover $S$.
- Remove $v$ and all edges incident to it.

Then return $S$.

This does return a valid vertex cover (because it continues until there's no vertices left), and it's polynomial in time. But it can have a really bad worst case.

**Example 18.21**

Suppose we have $k!$ vertices, with $k!$ vertices of degree $k$ at the top, and at the bottom $k!/k$ of degree $k$, then $k!/(k-1)$ of degree $k-1$, and so on, up to $k!$ of degree 1.

Initially, the blue vertices and the black vertices all have degree $k$. If we're unlucky, we might pick the two blue vertices; now all the black ones have a degree of $k-1$. If we're unlucky again, then we might pick all the green vertices. Now our black vertices have degree $k-2$. And so on.

So instead of getting a vertex cover just using the top row, we'll have a vertex cover using just the bottom row, which is really bad.

We can show that the vertex greedy algorithm is a $(\log n)$-approximation, which is worse than our algorithm from earlier.

*Proof.* Suppose that $|S_{\text{OPT}}| = m$. Then suppose we start with the entire graph and run an iteration of this algorithm of length $m$. We can use the same argument as in set cover to show that we can get a substantial fraction of the edges; and then we can bound the number of times we need to run through each vertex, and just as set cover this gets the boundn that we need. □

# §19 November 15, 2022 — Approximation Algorithms II

Today we will look at a technique based on linear programming to find approximation algorithms; we'll also look at two ways to approximate a special case of the travelling salesman problem.

## §19.1 Linear Programming Relaxation for Vertex Cover

We're essentially going to encode vertex cover as a linear program, and use the solution for linear programming to approximate the solution for vertex cover.

We have a graph with a bunch of vertices, and a bunch of edges; and we have to find vertices that we'll put into our cover. A useful approach is to have the variables in our linear program represent vertices — where we say $x_i = 0$ if $v_i$ is not in the cover, and $x_i = 1$ if it is in the cover. This gives a vertex-centric approach, where the variables indicate whether our vertices are in or out of the cover.

Starting here, we assign a LP variable $x_i$ to each $v_i \in V$; if $x_i = 1$ then we select $v_i$ and add it to $S$, and if $x_i = 0$ then we don't select $v_i$ for the cover.

Whether this is a useful place to start or not depends on whether we can encode the rest of the problem in the mathematics we're allowed to use in a linear program.

If we want to minimize the size of the cover, that's equivalent to minimizing $\sum_{i=1}^{n} x_i$.

Now we have to set up a set of constraints that ensure every edge in the graph is covered. Suppose we have an edge $v_iv_j$, so that its endpoints have values $x_i$ and $x_j$. For each such edge, we need at least one of $x_i$ and $x_j$ to be 1; in the language of linear programs, we can't ask for one of them to be 1, but we can add them together and ask that $x_i + x_j \geq 1$; if our assignments are all 0 and 1, then this guarantees one is at least 1.

We also may not be able to force the variables to be 0 or 1; but we can restrict then so that $0 \leq x_i \leq 1$.

This gives us the following linear program:

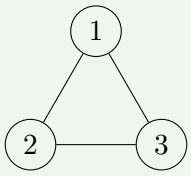
$$\begin{aligned}
&\text{minimize } \sum_{i=1}^{n} x_i \\
&\text{subject to } x_i + x_j \geq 1 \text{ for all edges } e = (v_i, v_j) \\
&\text{and } 0 \leq x_i \leq 1.
\end{aligned}$$

Before the last line, we'd written a *binary* integer linear program, where our variables had to be 0 and 1. There are not polynomial time solutions to those problems, but there are for *linear* programming. So what we've done is written our problem as a binary linear program, and then added the constraint $0 \leq x_i \leq 1$ to relax it and allow the $x_i$ to be continuous. This lets us use linear programming techniques to get our answer, and then somehow we need to map this back to what we need.

Let's suppose the optimal solution to the linear program is $x^*$. Now the problem is that it may contain non-integer values for some of the $x_i$.

**Example 19.1**

Suppose we had the following graph:




The optimal vertex cover is when we take $S = \{v_1, v_2\}$ for instance — corresponding to $x_1 = x_2 = 1$ and $x_3 = 0$. But the linear program will actually give us $x_1^* = x_2^* = x_3^* = 1/2$ — this satisfies the constraints, and gives a smaller value of 3/2.

Somehow, we need to 'round' these values that come out of the linear program to obtain integers; we'll use $x_i^{**}$ to denote how we do this.

One idea would be to take $x_i^{**} = 1$ only if the linear program gives us $x_i^* = 1$. But this doesn't work — in the above example, we'd round everything down to 0, and our approximation would be an empty set. This fails our condition that we'd like the approximation algorithm to give us a *valid* cover, even if it's not optimal in size.

To find a better idea, our linear programming solution guarantees that $x_i + x_j = 1$ for every edge. So then at least one of $x_i$ and $x_j$ is at least $1/2$. This suggests that we can round using $1/2$ as the cutoff — if $x \geq 1/2$ then we round up to 1, and if $x \leq 1/2$ then we round down to 0 — because the condition $x_i + x_j \geq 1$ guarantees this will give us a valid cover.

More formally, we set

$$x_i^{**} = \begin{cases} 1 & x_i^* \geq 1/2 \\ 0 & x_i^* < 1/2. \end{cases}$$

We're then setting

$$S = \{v_i \mid x_i^* \geq 1/2\},$$

and this is guaranteed to give us a valid cover.

How big is the cover? We have

$$|S| = \sum_{i=1}^{n} x_i^{**}.$$

In our approximation, we've changed the values to increase them, but this increase at most doubles them (and some are shrunk). So we have $x_i^{**} \leq 2x_i^*$, and therefore

$$|S_{\text{APX}}| \leq \sum_{i=1}^{n} x_i^{**} \leq 2 \sum_{i=1}^{n} x_i^*.$$

So our rounding gives a 2-approximation to the LP solution. (In our example, all the values were $1/2$, which round up to 1; this means our answer is 3, which is twice the LP value.)

But we have

$$\sum_{i=1}^{n} x_i^* \leq |S_{\text{OPT}}|,$$

by the way we defined the linear program — $x^*$ is the optimal point in the space of all solutions, and the allowed feasible solutions to the actual problem are the discrete points in the same space. So these can be no better than the actual optimum of the LP — one of those points is the discrete optimum, and this can be no better than the optimum in the relaxed space. This means

$$|S_{\text{APX}}| \leq 2\,|S_{\text{OPT}}|.$$

This is a general technique — the idea of taking a discrete problem and expressing it as a linear program, and using relaxation from the discrete to continuous case, and then obtaining an optimal solution and approximating to the nearest point in the integer space.

## §19.2 Travelling Salesman Problem

**Question 19.2.** In the travelling salesman problem:

- Given — $n$ cities $v_1, \ldots, v_n$ and the distances (costs) $d(i, j)$ corresponding to the cost from city $v_i$ to $v_j$.
- Decision version — Given a bound $D$, is there a tour that visits every city exactly once of cost at most $D$?
- Optimization version — Find a minimum length (cost) tour that visits every city exactly once.

The decision version is NP-complete, so the optimization version is NP-hard.

In the previous lecture, we saw constant valued approximations, in particular a 2-approximation. Here it's been shown that there's *no* polynomial time $c$-approximation algorithm for TSP for any $c > 1$, unless $\mathsf{P} = \mathsf{NP}$. So we can't even approximate this.

We also talked about polynomial-time approximation schemes. Similarly, there is no polynomial-time $f(n)$-approximation scheme for the general TSP for *any* function $f(n)$ that can be computed in polynomial time.

This is bad news, but there is some good news — even though approximating TSP in the general case is hard, approximating TSP when the distances are *metric* distances is not hard.

## §19.3 Metric Travelling Salesman Problem

In Metric TSP, the distances (costs) $d(i, j)$ satisfy the triangle inequality — for all $i$, $j$, and $k$ we have

$$d(i, j) \leq d(i, k) + d(k, j).$$

There are also some additional properties that metric distance should satisfy:

- $d(i, i) = 0$;
- $d(i, j) \geq 0$;
- $d(i, j) = d(j, i)$.

(This might not be true for airline tickets, for instance.)

This doesn't make the problem easy enough to be polynomial — metric TSP is still NP-hard. But it can now be *approximated* in NP time.

**Question 19.3.** Let $G$ be a weighted, *complete* graph, where each vertex coresponds to a city and each edge $(v_i, v_j)$ has a weight $d(i, j)$, with $d$ satisfying the metric space conditions.

Find a tour of minimum weight $D$ that visits each city exactly once.

The strategy we'll use (similarly to last lecture) involves two pieces — bounding the optimal and approximate solutions, to essentially the same thing (so that we can relate the approximate to optimal solution directly), without knowing what the optimal solution is.

More precisely, we want:

- A lower bound on the optimal cost by some function of the input;
- An upper bound on the approximation algorithm's cost, involving the same function.

We're going to do this by using MSTs. We'll do this twice — we'll find one approximation algorithm that gets a 2-approximation, and then we'll do it again with better bounds to get a $\frac{3}{2}$-approximation.

First, we need to find a lower bound on $D_{\text{OPT}}$. Let's call our MST $T$, and its weight $w(T)$. Then we have

$$D_{\text{OPT}} \geq w(T),$$

since if we start with the optimal tour (of weight $D_{\text{OPT}}$), we can turn it into a tree by removing an arbitrary edge; this tree has weight at most $D_{\text{OPT}}$, because our edge weights are nonnegative. But this is a spanning tree, and any spanning tree is bounded by the weight of the minimum spanning tree.

**Remark 19.4.** This is a very simple argument — we start with our optimal tour and remove any edge to get a spanning tree of weight at most $D_{\text{OPT}}$; this has weight at least $w(T)$, and we're done.

When you are stuck with an awkward explanation, you can often check whether there's an easier way of starting that gives a simpler explanation — because simpler explanations are often clearer.

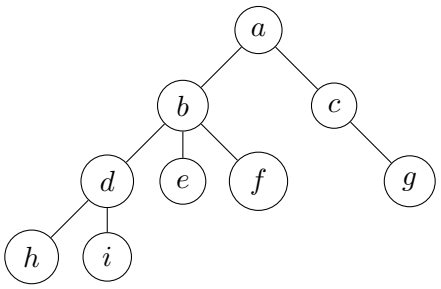
Now how do we turn this into an algorithm?

**Algorithm 19.5** (Double–Tree Algorithm) **—** We do the following:

1. Find the MST $T$ of our graph $G$.
2. Duplicate every edge in $T$.
3. Find an **Euler tour** on the multigraph formed by duplicating $T$.
4. Convert this Euler tour into a valid tour, by a process we'll call 'shortcutting.'

**Definition 19.6.** A **tour** in a graph is a path that begins and ends at the same vertex. An **Euler tour** is a path that uses every edge exactly once (we are allowed to revisit vertices).

As an example, suppose we start with the following MST:




One way of producing an Euler tour on the duplicated tree is by starting at one node and doing a depth-first search.

We now have the Euler tour ABDHDIDBEBFBACGCA. This is valid until we revisit D, which is bad; so we need to do something about that, to fix it.

What *shortcutting* does is instead of backtracking to D, we instead go directly from $H$ to $I$. The triangle inequality tells us that we haven't made the path any longer — so the length of the Euler tour serves as a bound for the length of the tour after the first shortcut. All the other shortcuts do the same thing — so when we're done shortcutting, our fianl tour — which is valid — is bounded by the length of the Euler tour. (In shortcutting, we essentially just cross out all vertices that we revisit, except for the thing at the root of the tree.)

**Student Question.** How do we know we have the edges needed for shortcutting?

What's drawn is just the MST, but underneath it is a complete graph — there has to be an edge there, because our graph is complete.

First, why can we always make this Euler tour?

**Theorem 19.7**

Any graph where all degrees are even has an Euler tour, and this Euler tour can be constructed in polynomial time.

If you look up 'bridges of Konisberg,' you will find interesting stories about the origins of graph theory.

So when we do this shortcutting, we'll end up with ABDHIEFCGA, which visits every vertex once.

The cost of the Euler tour is exactly twice the cost of the MST, since we traverse every edge twice; and the shortcutting only decreases our cost — so we have

$$D_{\text{DOUBLE-TREE}} \leq D_{\text{EULER}} = 2w(T) \leq 2D_{\text{OPT}}.$$

So then the double-tree solution is a 2-approximation.

There's a second, simlar, argument that's even better. It works similarly, creating an Euler tour and shortcutting it, but it creates a more efficient one:

**Algorithm 19.8** (Christofides Algorithm) — First we find the MST. Now we look at the MST and compute $O$, the set of all odd-degree vertices of $T$ (the size of $O$ must be even — since each edge contributes 1 to two vertices' degree, the sum of all degrees is even; if we remove the even-degree vertices, then we still have an even degree sum).

Now we find the minimum cost perfect matching $M$ of the subgraph induced by $O$. This can be computed in polynomial time (this is difficult to prove) — a matching is a set of edges witih no common endpoints.

Then we find an Euler tour of $T \cup M$, and we convert this Euler tour into a TSP tour by shortcutting.

**Example 19.9**

In our example, the odd-degree vertices are D, E, F, G, H, and I. We then take a minimum perfect matching of this; suppose it maps D with E, F with G, and H with I.

We didn't double edges, so now we only go over the original tree edges once. But we can now use this to produce an Euler tour (that uses every edge exactly once) ABDHIDEBFGCA. This may still hit some vertices more than once, but it's better than the other. Then we can do both shortcuts and get the final tour of ABDHIEFGCA.

Now we want to show that this is a 3/2-approximation.

The basic analysis is the same as before —

$$D_{\text{CHRISTO}} \leq w(T) + w(M) \leq D_{\text{OPT}} + w(M)$$

(the first inequality is again because the triangle inequality means shortcutting only helps us). To bound the approximation factor, we have to somehow bound $D_{\text{OPT}}$ in terms of $w(M)$.

**Claim —** We have $w(M) \leq D_{\text{OPT}}/2$.

*Proof.* Consider our optimal tour with cost $D_{\text{OPT}}$. Then we can draw our set $O$ on this tour in orange. Then we can shortcut the optimal tour into a cycle that only includes the odd-degree vertices; if we do this shortcutting, then we're left with an *O-tour* (a tour that just covers the vertices in $O$).

We know there's an even number of vertices in $O$, so we can choose alternate edges in the tour to create two sets $M_1$ and $M_2$, which both form matchings. Then we have

$$w(M_1) + w(M_2) = D_{\text{O-TOUR}} \leq D_{\text{OPT}}.$$

Meanwhile we have $w(M) \leq \min(w(M_1), w(M_2))$, so then $w(M) \leq \frac{1}{2} D_{\text{OPT}}$. □

Today we saw linear programming relaxation as one approach to approximating graph problems, and for metric TSP we saw wa general approach of creating an Euler tour, bounding it, and shortcutting; and we saw two ways of creating that Euler tour.

# §20 November 22, 2022 — Exponential Algorithms

## §20.1 Introduction

Today we will talk about *exponential* time algorithms. These algorithms don't run in polynomial time, or even slightly superpolynomial time — they run in exponential time. So we can wonder why they're interesting at all. The simplified version of what we try to convey in algorithms courses is that polynomial is good and exponential is bad, so why would we discuss exponential ones?

But even if an algorithm is exponential, it matters a lot *how* exponential it is. So even within exponential, there could be a whole range of runtimes — some exponential algorithms can actually be pretty good for the problems we want to solve. And often this is important in practice.

## §20.2 Dealing with Hard Problems

**Question 20.1.** What do you do when the problem you want to solve is NP-hard?

Last two classes, we saw *approximation algorithms*, which run quickly but might not find the right answer — if the optimal subcover is of size 100, we might find one of size 200. These are often useful because they run in polynomial time, and the difference between 100 and 200 may not matter that much.

Today we will look at *exponential* time algorithms which correctly find the optimal solution, but are allowed to run in exponential time. Nevertheless, these algorithms are very useful, especially if our problems are not that big.

**Question 20.2.** In which situations would it be worth it to run exponential algorithms to get the exact solution, rather than approximation algorithms that give an approximate solution?

There are some problems because it's worth putting lots of computation power in to solve the problem to optimality. Some of the most well-known examples are when we want to optimize something that has a large number of money involved. For reasons Prof. Indyk doesn't fully understand, people aren't happy if they get half as much money as they could; so people want to invest more computation to get better outcomes.

We'll see other examples with decision problems — sometimes solving SAT correctly is important for security.

There are also other ways of dealing with hard problems — parametrized complexity and average case complexity.

## §20.3 Meeting in the Middle

Our first technique that we'll use to get better exponential time algorithms is called the *meet in the middle* technique. It's very popular in complexity theory, optimization, and especially cryptography (very often you can use it to break the security of systems, and we'll soon see why).

On a high level, it reduces runtime from $2^n$ to $2^{n/2}$ — it cuts what we have in the exponent by a factor of 2. What this means is that we can solve twice as big instances in the same time. So in cryptography, if before we could break 40-bit encryption, after using this technique you could now break 80-bit encryption.

We've already seen a glimpse of this technique on the first quiz:

**Example 20.3**

Given three sets $A$, $B$, and $C$ of size $n$, check if there exist $a \in A$, $b \in B$, and $c \in C$ with $a + b + c = 0$.

The idea of the efficient solution is to first enumerate all pairs $(a, b)$, compute and store their sums in some hash table, and then for all $c \in C$, check if $-c$ has been stored in the hash table.

Pictorially, this algorithm splits our numbers in the middle, between $(a, b)$ and $c$. On one side we enumerate all possible answers, and on the other side we also enumerate all potential candidates; and then we check if the two sides meet (meaning that the numbers on the left and right match).

This algorithm runs in $O(n^2)$; this is significantly better than the naive algorithm that enumerates all triples $(a, b, c)$ and checks if they sum to 0.

**Remark 20.4.** The best algorithm known is $n^{2-o(1)}$, so this is essentially optimal.

Now let's see how this applies to an actual NP-hard problem.

### §20.3.1 Subset Sum

**Question 20.5.** SUBSET-SUM is the following problem:

- GIVEN — a set of $n$ integers $S = \{a_1, \ldots, a_n\}$ and an integer $t$.
- GOAL — check if there exists a subset $A \subseteq \{1, 2, \ldots, n\}$ such that $\sum_{i \in A} a_i = t$.

This problem is NP-hard. It is a very natural and useful problem, and it has connections to other problems (like knapsack) as well as cryptography.

The naive solution would be to follow the definition directly — we can enumerate all the subsets, of which there are $2^n$. Then for each subset that we enumerate, we can sum up all its elements, and see if they sum up to $t$. In the natural interpretation, the runtime is $O(n \cdot 2^n)$ (since it takes $O(n)$ operations to check the sum of each set).

If $n$ is not too large, then this can actually run pretty efficiently. But we can do better using the meet in the middle technique.

We can split our set $S$ in half — equivalently, we can decompose $A$ into two subsets $A_1 \subseteq \{1, 2, \ldots, \lfloor n/2 \rfloor\}$ and $A_2 \subseteq \{\lfloor n/2 \rfloor + 1, \ldots, n\}$ such that $\sum_{i \in A_1} a_i = t - \sum_{i \in A_2} a_i$. It's clear that this is an equivalent formulation (we essentially split $A$ into a left side and a right side, and call the left set $A_1$ and the right set $A_2$; conversely if we're given $A_1$ and $A_2$, we can merge them to obtain $A$).

Now we can split this formulation in the middle, so we enumerate all $\sum_{i \in A_1} a_i$ and all $t - \sum_{i \in A_2} a_i$, and then we try to match them.

To implement this idea, we first enumerate all sets $A_1$, and for each set we store $\sum_{i \in A_1} a_i$ using hashing. Meanwhile, for each stored set $A_2$, we check whether $t - \sum_{i \in A_2} a_i$ has been stored in the hash table. If the lookup is satisfactory, then we have a match between some set $A_1$ and $A_2$, which means there is a pair satisfying this equality and the answer to the subset sum problem is Yes.

**Remark 20.6.** Here we are splitting the *underlying* input, with $n$ numbers, into two halves of size $n/2$. In the $a + b + c$ problem, we didn't exactly split the input, but the expression. So the implementation is a bit different, but the philosophy is the same.

Now let's analyze the runtime. It takes $O(n2^{n/2})$ time to enumerate all sets $A_1$ and compute and hash their sums. (There's $2^{n/2}$ sets $A_1$, and we enumerate all of them and sum all the elements in the set.) The same runtime analysis holds for $A_2$ — we enumerate $2^{n/2}$ sets, sum them, and look them up in the hash table. This means the total number of operations we have to perform is $O(n \cdot 2^{n/2})$ (including both arithmetic operations and hashing). What we've won is that now we have $n/2$ in the exponent, as opposed to $n$.

**Student Question.** *Is this average-case analysis?*

**Answer.** It's randomized, since in hashing we're using random bits. So this gives an *expected* runtime.

There is another caveat — we took $O(n \cdot 2^{n/2})$ operations, where an operation could be addition, subtraction, or whatever we need for hashing. But with problems like this, sometimes the numbers in principle could be very large. So we could potentially have issues if the numbers we're dealing with have cryptographic size — can we add $2 \cdot 10^5$-digit numbers in constant time? If our operations themselves took superconstant time, we'd have to incorporate that as well. But this wouldn't change the complexity that much — it might give us an extra factor of $n$, but it won't change the overall situation in the exponent.

We could slightly change this algorithm to not use randomization, by replacing hashing with something else: for each $A_1$, we generate a pair $(\sum_{i \in A_i} a_i, \text{red})$, and for each $A_2$, we generate a pair $(\sum_{i \in A_i} a_i, \text{blue})$. We then sort all pairs by the first element. Then we have a match if and only if we have two consecutive pairs in the sorted order with the same first argument, and different color.

The complexity is the same — the complexity of the first step is still $n \cdot 2^{n/2}$, while sorting can be done in $n \log n$ time, which here is $2^{n/2} \log 2^{n/2} = O(n \cdot 2^{n/2})$ as well.

**Student Question.** *Is this worse in terms of constant factors?*

**Answer.** Not sure — it depends on how efficiently it's coded. There are fairly efficient subroutines for sorting, so it shouldn't take that much time.

**Student Question.** *What would happen if we split the subsets in half?*

**Answer.** Unfortunately, this technique isn't very amenable to recursion — the problem is that things are asymmetric, where we're storing one side and enumerating the other side. In the vast majority of cases you can't really recurse in this way — one way of thinking about it is that if you recurse too often, then you might be able to solve it in polynomial time, and you'd have proven that P = NP (which means chances are that it won't work).

**Remark 20.7.** One problem with this algorithm is that it improves the time, but it uses exponential space; and space is often a bigger bottleneck than time. People have studied algorithms which are efficient in terms of time and don't use that much space. In 1979 Schroeppel–Shamir found an algorithm with roughly $O(2^{n/2})$ time and $O(2^{n/4})$ space. Very recently, this has been improved to $O(2^{0.249999n})$ space. More practically, we can get a $O(2^{0.86n})$-time algorithm that uses *polynomial* space (which matters because the limitation for space is often more severe than for time).

## §20.4 Branch and Bound

We'll now move on to the second technique, which is called *branch and bound*. Like meet in the middle, it's a very powerful and general technique. We'll see applications to satisfiability, but it's also useful for integer programming and other problems.

### §20.4.1 3-SAT

**Question 20.8.** In 3-SAT:

- GIVEN — a formula $\varphi$ with $m$ CNF clauses, each with at most 3 literals, over $n$ Boolean variables.
- GOAL — check whether the formula is satisfiable.

**Example 20.9**

An example of 3-SAT would be the formula

$$(x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_3} \vee x_4) \wedge (\overline{x_2} \vee \overline{x_3}).$$

This is satisfiable — we can set $x_1 = x_2 = 1$ and $x_3 = x_4 = 0$.

We know that 3-SAT is NP-complete, so we probably can't solve it in polynomial time. But people still want to solve it. For example, there are practical industrial applications of 3-SAT — for example, you can use them to design locks so that one key can open a certain subset, and so on. Or for example, we have specifications for our code and want to verify it's correct. Or if we have a circuit, and we want to verify that the circuit doesn't make errors (which unfortunately it occasionally does — 10 years ago there was a problem with addition in Intel which was a big deal). Any sort of verification that a circuit or lock system does what it's supposed to do, these days, is done by expressing it in 3-SAT (or more generally SAT) and running SAT solvers.

So despite the fact that satisfiability is NP-complete, it's very widely used in practice. Of course the SAT solvers are still subject to NP-hardness, but because they're not naively enumerating, they're often good enough to still use in practice.

**Remark 20.10.** The reason 3-SAT is special is that it's NP-hard, but 2-SAT can be solved in $O(m+n)$. This algorithm uses strongly connected components, which we haven't seen in a while, so we won't go over the proof; but this is the reason we talk about 3-SAT and not 2-SAT.

We have a naive algorithm for 3-SAT that takes $O((m+n)2^n)$ time — enumerate all possibilities and check if every clause is happy. Now we'll see an improvement using branch and bound, by Monien–Spekenmeyer 1985.

First, we can assume there's a clause $\ell_i \vee \ell_j \vee \ell_k$ with three different variables — otherwise the problem would be 2-SAT, which we can solve in polynomial time.

Consider a clause with three distinct variables $\ell_i \vee \ell_j \vee \ell_k$. Then there are three situations in which $\varphi$ is true:

- $\ell_i$ is true;
- $\ell_i$ is false but $\ell_j$ is true;
- $\ell_i$ and $\ell_j$ are false, but $\ell_k$ is true.

It's clear that these cases are disjoint and exhaustive (the only alternative is that everything is false, but then the clause is not satisfied, so the whole formula is not satisfied).

**Student Question.** *Do we have to consider when more than one literal is true — like if $\ell_i$ and $\ell_j$ is true?*

**Answer.** This would fall under either case 1 or case 2. If we enumerated all the combinations of the three literals which satisfy the clause, then there would be 7 cases, but we've factored some of the cases together — the case where $\ell_i$ is true involves all four combinations of the other two variables, and similarly the case where $\ell_i$ is false and $\ell_j$ is true includes both cases of $\ell_k$.

We don't know which case we're in — if we knew, then we could keep setting things and solve the problem. So our algorithm is going to have to consider all three cases to be possible.

**Algorithm 20.11 —** The algorithm is as follows:

- Find a clause $\ell_i \vee \ell_j \vee \ell_k$ with three distinct variables. (If no such clause exists, solve 2-SAT.)
- Set $\ell_i$ to true, and recurse on the remaining $n-1$ variables.
- Set $\ell_i$ to false and $\ell_j$ to true, and recurse on the remaining $n-2$ variables.
- Set $\ell_i$ and $\ell_j$ to false and $\ell_k$ to true, and recurse on the remaining $n-3$ variables.

**Student Question.** *Does the algorithm go through all three cases?*

**Answer.** Yes, since we don't know which case our answer will be in.

Of course, if at some point in the recursion we end up with 2-SAT instead of 3-SAT, we solve it in polynomial time.

**Student Question.** *In the first two cases, we don't eliminate the other variables in the clause?*

**Answer.** We don't eliminate the variables, but we can sometimes simplify clauses. For example, we can take the formula

$$\varphi = (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_3} \vee \overline{x_4}) \wedge (\overline{x_2} \vee \overline{x_3}).$$

Let's suppose the clause we branch on is $x_1 \vee x_2 \vee x_3$. In the first case we set the first literal to true, meaning $x_1 = \mathsf{True}$. Then we recurse on the remaining 3 variables — we can forget about the first clause (which is now satisfied). The second clause is actually satisfied as well, so we can recurse on the clause $(\overline{x_2} \vee \overline{x_3})$. This is actually a 2-SAT clause, so we don't have to recurse anymore.

So we always discard the original clause, and we can also make a few other simplifications. For example, in the second case where $x_1 = \mathsf{false}$ and $x_2 = \mathsf{true}$, then we remove the first clause (which we have satisfied), we can simplify the second clause to $(\overline{x_3} \vee \overline{x_4})$ (since $x_1$ is not going to help us), and we can simplify the third to $\overline{x_3}$ (since $\overline{x_2}$ is false so is not going to help us).

**Student Question.** *Aren't we recursing on $n-3$ variables no matter what?*

**Answer.** In the first two cases, we're not getting rid of the other variables, since they can appear in other clauses as well.

**Student Question.** *Why is this called branch and bound?*

**Answer.** Some of the names are just historical, and were originally applied in a somewhat different context. But we can think of the recursion as branching, and we can think of the constraints as bounding (it may be better to think of it as 'branch and constrain'). This technique is popular in integer programming, where bounding means that the solution lies on some side of a certain hyperplane.

Now let's analyze runtime — does this really save us anything? It turns out the answer is yes.

Let $T(n, m)$ be the runtime of the algorithm. The number of clauses is not really important in this recurrence, and we can just ignore it; the $n$ is a different story.

We have

$$T(n, m) \leq c(n + m) + T(n - 1, m) + T(n - 2, m) + T(n - 3, m),$$

since we could potentially be doing something in linear time (e.g., scanning for clauses, or solving 2-SAT). This is not a pretty recurrence, and we can't apply the master theorem to it. But we could instead guess an answer — suppose that $T(m, n) \leq a^n(n + m)$ for some constant $a$ (which we will hope is less than 2). Then using the inductive assumption, our right-hand side is bounded by

$$(c + a^{n-1} + a^{n-2} + a^{n-3})(m + n) = a^{n-1}\left(1 + \frac{1}{a} + \frac{1}{a^2} + \frac{c}{a^{n-1}}\right)(n + m).$$

We hope to show that this is less than $a^n(m + n)$ for some $a < 2$.

We can ignore the $m + n$; the real action is happening in the first factor. We will make our life easier by dropping $c$, because as $n \to \infty$ the last term tends to 0 (so it's not going to matter in the calculation; it's fairly easy to account for it in the end). Then we want to have

$$1 + \frac{1}{a} + \frac{1}{a^2} \leq a$$

for some $a < 2$ (by dividing out by $a^{n-1}(m + n)$). This is definitely possible, because if $a = 2$ then we have $1 + 1/2 + 1/4 = 7/4 < 2$, which means we have some slack. We can verify this by plugging in numbers — for example $a = 1.9$ works. So this demonstrates we can get an algorithm with $1.9^n$.

We can actually solve the cubic equation, which gives $a \approx 1.84$. So instead of a $2^n$ algorithm, we get a $1.84^n$ algorithm, which is an improvement — this is polynomially faster than $2^n$.

So now we have an algorithm with runtime $O((m + n)1.84^n)$. This algorithm was discovered in 1985, but there's lots of work being done on this problem. The state of the art is $O((m + n)1.308^n)$, which is quite a significant improvement. This can in fact handle serious formulas — if $n = 100$, then we have $1.308^{100} \approx 4 \cdot 10^{11}$. Processors these days are able to handle such sizes. In contrast, $2^{100}$ is huge; even if you have a very fast processor, you cannot execute this. Essentially, 1.308 allows you to solve problems a factor of 3 larger.

## §20.5 Conclusions

So far, we've seen two techniques for dealing with hard problems. Another technique is *parametrized complexity* — where our algorithms are exponential in some parameter, as opposed to the input size. In SAT, people observed that in practice, the formulas trying to be solved often have properties that can be expressed in terms of other parameters.

Combining paramterized complexity and these efficient algorithms, we can get good SAT solvers for the cases which are relevant in practice — there are naturally occurring instances of satisfiability with a million variables that SAT solvers can solve. This does *not* mean SAT solvers can solve all problems with a million variables — you can express RSA using a few thousand variables, but it's not broken. It means there are certain naturally occurring instances of SAT that are really large and that we can solve, but there are no known algorithms that can solve *every* SAT formula of this size.

**Conjecture 20.12** (Exponential Time Hypothesis) — Any algorithm for 3-SAT takes $2^{\Omega(n)}$ time.

Note that this is stronger than $\mathsf{P} \neq \mathsf{NP}$. This conjecture is generally believed to be true, despite the fact that for some instances SAT solvers can run very efficiently.

**Student Question.** *Can we use SAT solvers and reductions to solve other NP-hard problems?*

**Answer.** Yes — this is done fairly often. It doesn't work always (just because something is reducible to 3-SAT doesn't mean SAT solvers can handle the hard instances of such problems), but it's quite typical that problems you encounter in practice have some structure that makes them doable. In particular, SAT solvers have a hard time breaking cryptography because those formulas are hard instances. So there's an interesting situation where just because something's NP-hard doesn't mean you can't solve specific cases.

# §21 Novembe 29, 2022 — Online Learning: Multiplicative Weights Algorithms

In multiplicative weights algorithms, we combine advice from different models and try to find the right way of combining them to make a good decision. The same family of approaches is useful for a broad variety of problems — to determine approximate solutions to linear programs, to find Nash equilibria in two-player zero-sum games, as a basis for approximate algorithms for many combinatorial optimization problems, and for boosting in machine learning (where there's a variety of machine learning models and we want to combine them to make the best overall prediction).

Imagine the scenario where you're a stock trader, and you have a panel of experts $E_1, \ldots, E_n$. Every morning, these experts give you binary advice on whether they think the stocks will go up or down. You have to make a decision by combining their advice; then you do your trade, and at the end of the day you find out what happened — whether your decision was correct, and which of your experts were correct. So you get daily feedback on mistakes — let $m_1^{(t)}, \ldots, m_n^{(t)}$ be whether a mistake was made by expert $i$ on day $t$ (which is a binary variable), and let $m^{(t)}$ be whether your overall decision was a mistake (so these are binary variables — 1 for a mistake and 0 for a correct decision).

**Question 21.1.** How can we define an algorithm to combine their advice, and can we bound how often we'll be wrong?

This is an *online* problem — I have to make a decision after every input, and before I get feedback. We'll assume nothing about the pattern of the stock. We're going to use words like 'maybe one of the experts is better than the rest and we want to focus our attention on the better expert,' so we might imagine that the experts may differ in accuracy. In the first instance we'll look at, this will be true; but in the end, we won't assume *anything* about the quality of the experts either.

**Question 21.2.** How can we take into account the advice we get from the experts, and the feedback we get?

We probably want to combine their advice in some weighted way. And we may want to reduce the weight of someone who's frequently wrong — if an expert is wrong four days in a row, we may not want to believe them as much.

SO we can imagine first believing everyone equally and taking a vote, and then the next day we imagine the correct people are somewhat more dependable and the incorrect ones are less — so some way of discounting the experts who deliver wrong information makes intuitive sense.

We'll start with this idea, but in the end we won't actually assume any consistency in advice (i.e. that someone who gives good advice in the beginning will give good advice at the end).

**Question 21.3.** What should we compare ourselves to when evaluating our algorithm?

One idea is to say that the best we can do is perfect, so we can count the number of mistakes; if this is bigger than 0, that's not as good. So one idea is to treat no mistakes as the gold standard.

When we did competitive analysis, what we said there was that we compared ourselves to the cost that the best offline algorithm would get, where an offline algorithm gets to see all future as well as past inputs. For us, one of the inputs is the feedback; so if we could see that, then we'd know which expert was most accurate, and we could believe that expert and ignore everyone else. Or we'd know what the right answer was, since we get that feedback. So that corresponds to what we did in competitive analysis.

But the problem is that we don't control the experts, only the algorithm — we can only improve the algorithm, but we are stuck with the experts. So if we want a measure asking how good is the algorithm given the advice we started with, maybe we don't wnat to compare ourselves to no mistakes, but rather to the base of the experts. So the idea we are going to use is that *the best expert (the expert who makes the least mistakes) is the gold standard.*

The reason to use this is so that it's a way of not penalizing our algorithm if the experts are bad.

Recall that we define our 'mistake' variables for the expert $i$ on day $t$ as

$$m_i^{(t)} = \begin{cases} 1 & \text{expert } i \text{ makes a mistake on day } t \\ 0 & \text{expert } i \text{ does not make a mistake.} \end{cases}$$

**Definition 21.4.** Our **regret** is $\sum_{t=1}^{T} m^{(t)} - \min_{i\in[n]} \sum_{t=1}^{T} m_i^{(t)}$ — the difference between the number of mistakes we made and the number of mistakes the best expert made.

**Definition 21.5.** We let $M^{(T)} = \sum_{t=1}^{T} m^{(t)}$, and we define $M_i^{(T)}$ similarly.

This is usually 0 or positive, but it could actually be negative — if we have a panel where most people are right most of the time, we might be able to do better than most experts.

**Example 21.6**

Imagine there's one expert who's perfect — who gives the correct prediction every single day. We can imagine the algorithm where we take feedback from everyone and fire everyone who made an incorrect decision — everyone who's remaining votes every day, and eventually you'll only have the perfect expert left and you'll never make a mistake again.

Every time you make a mistake, you'll fire at least one expert. So you are not going to make more than $N-1$ mistakes. In fact, every time you make an incorrect decision, you fire a lot *more* experts than just 1. So we can upper-bound the number of incorrect decisions we make.

We need some mathematical relationship that's inherent in the problem — that's like the leverage we use to get to the next step.

In this case, whenever we make a wrong decision, we fire at least half the experts. In fact, we're ignoring part of the algorithm here, since we may fire people even when we make the correct decision.

So we can let $S_1$ be the set of experts on the first day, which is $\{E_1, \dots, E_n\}$; so $|S_1| = n$. Then we can let $S_t$ be the set on day $t$. We know that

$$|S_{t+1}| \leq \frac{1}{2} |S_t|$$

if $m^{(t)} = 1$.

Once we get down to one person, that's the perfect expert, and we never make any mistakes after that. Every time we make a mistake, we cut the number of experts in (at most) half. So whatever size we started with as $|S_1|$ will get down to 1 after $\log_2 n$ halvings, or mistakes.

So after at most $\lceil \log_2 n \rceil$ mistaken decisions, the size of the set becomes 1, and then there's no more mistakes.

> **Student Question.** HOw are we making the decision?
>
> Majority vote of the remaining experts.

This is a helpful analysis, but it's not very realistic; there is no guarantee there's a perfect expert not on your panel. So how can we adapt this to a case where no expert is known to be perfect?

One way we could do this is run the algorithm until we have no experts left. And at the end of that day, we hire back all the exerts, and start again. You'll find that you can bound that by $M^* \log_2 n$, where $M^*$ is the cumulative number of mistakes made by your best expert.

One feature of this is that we said we would start by changing how much we believe an expert based on their accuracy. Here we have done that, but we have done the super extreme of that. So what if we do all of this again, but we don't have the penalty be to change their weight to 0 when they're wrong, but instead have a softer penalty?

**Algorithm 21.7** (Weighted Majority Algorithm) — For each expert, maintain a dynamic weight $w_i^{(t)}$, which we want to somehow reflect the accuracy. On the first day $t = 1$, we set all weights $w_i^{(1)}$ to 1. On future days, we update the weights based on incorrect predictions — to do this, we multiply the weights corresponding to incorrect experts by $1 - \varepsilon$ (where $\varepsilon$ is a parameter of the algorithm), and leave the weight unchanged otherwise. To aggregate, we take a weighted average majority vote.

We would love to be able to bound the number of mistakes this algorithm makes. It turns out that we can:

**Theorem 21.8**

Let $0 < \varepsilon \leq 1/2$, and let $M_i^{(T)}$ be the cumulative number of mistakes made by expert $i$ over the first $T$ days, so that $M_i^{(T)} = \sum_t m_i^{(t)}$. Let $M^{(T)}$ be the cumulative number of mistaken decisions you make over the first $T$ days. Then

$$M^{(T)} \leq 2(1 + \varepsilon) M_i^{(T)} + \frac{2 \log n}{\varepsilon}$$

for all experts $i$.

There's two things to note:

**Remark 21.9.** This is true for $M_i$ being any expert, so we might as well pick the best expert here; then this bounds the number of bad decisions we make by the number of mistakes made by the best expert, which is what we wanted as our gold standard.

Note that for very long times, we might imagine that even the best expert continues to make mistakes, so the first term will eventually dominate the second term — the effect of the term $2\log n/\varepsilon$ diminishes over time.

So if we imagine a very small $\varepsilon$, this tells us the number of mistakes the algorithm makes is bounded by around twice the number of mistakes the best expert makes.

*Proof.* Let's first look at the weight of an expert at some point in the algorithm, at a time $t$; this is $(1-\varepsilon)^{M_i^{(t)}}$. It's useful to also have the total weight of the experts at that time, which is

$$W(t) = \sum_{i=1}^{n} W_i^{(t)}.$$

Note that this total weight on the first day is just $n$.

When you make a mistake, at least half your weight made the mistake, and wrong decisions get multiplied by $1-\varepsilon$. And so that means at least half your weight gets multiplied by $1-\varepsilon$.

So when the algorithm makes a wrong decision, the weighted majority *produced* that wrong decision — that must have been at least $W^{(t)}/2$.

And that wrong portion of the panel is reweighted by multiplying by $1-\varepsilon$. And so the total weight of all the experts when the mistake is made is multiplied by a factor of — at most half of the weight got it right, and they are reweighted by 1, while the other portion that's at least half gets multiplied by $1-\varepsilon$. So then the total weight gets multiplied by a factor of at most

$$\frac{1}{2}\cdot 1 + \frac{1}{2}\cdot(1-\varepsilon) = 1 - \frac{\varepsilon}{2}.$$

(If more experts are wrong, then the factor gets even smaller.) So this means

$$W^{(t+1)} \le \left(1 - \frac{\varepsilon}{2}\right) W^{(t)}$$

when an incorrect decision was made. And this means

$$W^{(T)} \le W^{(1)} \cdot \left(1 - \frac{\varepsilon}{2}\right)^{M(t)}.$$

Meanwhile, for each expert $i$, their weight after $T$ steps is $(1-\varepsilon)^{M_i^{(T)}}$. And the weight of this one expert is at most the sum of the weights of all the experts — we have

$$(1-\varepsilon)^{M_i(T)} \le \sum_j (1-\varepsilon)^{M_i(T)} = W^{(T)} \le n \cdot \left(1 - \frac{\varepsilon}{2}\right)^{M(t)}.$$

Taking ln of both sides, we get

$$M_i^{(t)} \log(1-\varepsilon) \le \log n + M(t) \log\left(1 - \frac{\varepsilon}{2}\right).$$

Now we can divide by $\log(1-\varepsilon/2)$ (noting that this is negative), to get

$$M^{(t)} \le M_i^{(T)} \cdot \frac{\log(1-\varepsilon)}{\log(1-\varepsilon/2)} - \frac{\log n}{\log(1-\varepsilon/2)}.$$

Using a Taylor series expansion, we can get that

$$\varepsilon \leq -\log(1-\varepsilon) \leq \varepsilon + \frac{\varepsilon^2}{2}$$

(the second term is true for $\varepsilon \leq 1/2$). This then gives us a bound of

$$M^{(t)} \leq M_i^{(t)} \frac{\varepsilon + \frac{\varepsilon^2}{2}}{\frac{\varepsilon}{2}} + \frac{\log n}{\frac{\varepsilon}{2}},$$

which is what we wanted to prove. □

The heart of this was sort of the same as the case with the perfect expert — we looked at what happened when the algorithm amde a wrong decision, and propagated this into a bound involving the number of mistakes made by the algorithm and the number of mistakes made by the best expert.

It's helpful to think about — the algorithm makes roughly twice as many mistakes as the best expert in the worst case.

**Exercise 21.10.** If you were the adversary and you were designing the input, what would you do to get that worst case?

This algorithm we used generalizes:

**Algorithm 21.11** (Multiplicative Weights Update Algorithm) — Suppose that instaed of binary decisions and outcomes and mistakes, we have *continuous* ones. In particular, the mistake function will be continuous — $m_i^{(t)} \in [-1, 1]$, and a more negative value is associated with a better performance. In place of voting, we're going to randomize selection of one expert — every day, we pick one expert and go with whatever they say. But we'll pick them randomly from a weighted distribution, based on their past performance. This does two things. First, it changes our performance metric to an *expected* performance. But it also substantially improves the performance of the algorithm — expert $i$ is picked at time $t$ with probability

$$p_i^{(t)} = \frac{W_i^{(t)}}{W^{(t)}},$$

where the initial weights are all 1, and we update by

$$w^{(t+1)} = e^{-\varepsilon m_i(t)}$$

(this goes down if $m$ is positive and up if $m$ is negative).

**Theorem 21.12**

For the above algorithm (including the exponential reweighting), for some $\varepsilon \leq 1$, for any expert $i^*$,

$$\sum_{t=1}^{T} \sum_{i=1}^{n} p_i^{(t)} m_i^{(t)} \leq \sum_{t=1}^{T} m_{i^*}^{(t)} + \frac{\ln n}{\varepsilon} + \varepsilon T.$$

This is an interesting bound because we have our expected performance, the performance of the best expert, and the rest is not much — a constnat not depending on the number of cycles, and a small constant depending on the number of cycles. So then you cna make the time-average regret by moving the $m_{i^*}$ to the left and dividing by $T$; then what's left is the first term over $T$, and $\varepsilon$. This is called 0-regret, since in large time the first thing goes to 0, and you can make $\varepsilon$ as small as we want.

# §22 December 1, 2022 — Sketching and Similarity Search

Today we'll focus on a common problem that amounts to searching for useful objects in a large collection of data. As we all know, there's a lot of data out there; having tools that make it possible to search for useful data in this big sea is very useful.

This will be encapsulated by a *similarity search*, which we will formally define soon. Equally interesting will be the technique we'll use to obtain our algorithm, which is the technique of *sketching* — in a nutshell, it's a way to take our data and compress it (potentially by a lot) while still preserving certain properties. Then we can perform computation on this much shorter representation instead; this will save not only time but also space (as we will see next week).

Today we will see how to compress data such that we can still solve our problems on the compressed data, with much smaller time overhead.

## §22.1 Similarity Search

Our problem is very broad, and has many applications — so it has many formulations. For today, we will focus on one particular formulation (a data structure problem), which has the property of being self-contained but also very broadly applicable.

**Problem 22.1.** Suppose we have $n$ sets $A_1, \ldots, A_n$, which are all subsets of some universe $U$. We are also given a number $s \in [0, 1]$, which we refer to as the **similarity threshold**. Our goal is to build a data structure such that later, if we are given some set $A \subseteq U$, the data structure can check whether there exists at least one set $A_i$ such that the similarity between $A$ and $A_i$, denoted by $J(A, A_i)$, is at least $s$.

Pictorially, we have a database consisting of a large collection of sets $A_1, \ldots, A_n$ (we are given this database in advance, and we can preprocess it or do whatever we want with it). Later, the user comes with another set. Our goal is to detect whether there is at least one set in our database which is similar to the query set.

Of course, we need to define the similarity. There are many different ways to define the similarity, but today we will use the following:

**Definition 22.2.** The **Jaccard similarity** between two sets $A$ and $B$ is

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}.$$

(This is why we use the notation $J$.)



Pictorially, the notion of similarity measures how big the overlap between $A$ and $B$ is, as a fraction of the total space in the picture. Note that if $A$ and $B$ are equal, then their similarity is 1; so the highest similarity

we can get is when the two sets are equal (which makes sense). On the other hand, if the sets are completely disjoint, then $A \cap B$ is empty, so the similarity is 0. So equal sets have high similarity and disjoint sets have zero similarity.

This gives us one formalization of similarity search (there are many others).

In most applications, the sets $A_i$ are not arbitrary, and have size up to some parameter $d$. This is useful to keep in mind because the universe $U$ could be huge — if the sets are sets of words, then $U$ is the set of *all* words, which is enormous. Usually the sets $A_i$ are nowhere near as big, and it's useful to have a parameter that upper bounds the size of these variable sets.

To recap, our problem is to check whether there exists a set $A_i$ such that

$$J(A, A_i) = \frac{|A \cap A_i|}{|A \cup A_i|} \geq s.$$

**Remark 22.3.** One example application is searching for similar text documents (for example, if we have a legal document and want to search for similar ones to understand the background legal situation). The database then contains all the legal documents in the world, and each set is a set of words or phrases in the $i$th document. In applications we can skip words that are not important (such as 'the' or 'a') and only keep the words or phrases which are important; then we can represent each document as a set of words or phrases. Then given a new document, we want to find documents similar to the one we are working with.

In this situation, typically the data is very big — if the documents we are searching for are webpages, then $n$ can be in the billions. These documents themselves can also be pretty large (consisting of thousands of words).

**Remark 22.4.** Another application from biology is searching for similar genetic sequences. We may have a large collection of genetic sequences, and we can represent a sequence by its $k$-mers — the set of all substrings of length $k$ in our sequence. Then two sequences are similar if they contain similar $k$-mers.

**Remark 22.5.** Technically, our problem is not required to actually report the $A_i$; it is fine to just detect whether such an $A_i$ exists. We just formulated it this way to make things simpler; but of course in applications we want to find the set $A_i$. (If we are a biologist, our boss won't pay us to say that we found something but don't know what it is; we want to report the similar things.) But the algorithms we'll see do also solve the reportingn problem; the formulation just becomes more complicated.

**Student Question.** *Is a detection problem like a decision problem?*

**Answer.** Yes. This is a good analogy — when discussing NP-hardness, we converted optimization problems to decision problems because they were easier to deal with. That's similar to what we're doing here — the underlying problem is a search problem, but for the ease of exposition here we'll focus on the decision (or detection) problem.

**Student Question.** *Do the $A_i$ have to be disjoint?*

**Answer.** No, the sets are arbitrary. Usually they won't be; if our sets are legal documents, the word 'legal' may appear in all of them.

## §22.2 Baseline Solution

If we simply wanted to code something up without worrying about efficiency, the definition lends itself well to a linear scan (or for loop) — we can compute $J(A, A_i)$ over all sets $A_i$ (from the first set to the last), and for each set, check if any of the similarities is at least $s$.

This is the simplest solution to the problem — it's a single loop. But it is not very efficient. The loop runs over $n$ elements, and to compute the similarity between two sets, we have to go over the sets and identify elements belonging to both. We can do this using hashing — we compute a hash table for the union, and elements hashed twice occur in the intersection. So it's possible to check similarity in time linear in the set size. Since the sets have size at most $d$, and there are $n$ sets, the total time is $O(nd)$. This is generally pretty bad — in the applications we discussed, $n$ could be in the billions, and $d$ could be a thousand or more. If we multiply them, then we're getting pretty large numbers. You could do this without problems a few times, but if you're a web server doing this, you want to do it in microseconds; in situations like that, $nd$ is a very bad solution. (For example, what would happen if you replaced all hash tables in the world by lists? This would be a similar kind of slowdown.)

So this solution is simple but inefficient. What we will see in the remainder of the lecture is two ways of improving this simple solution. The first method will reduce the dependence on $d$ — we will still do a linear scan of all the sets, but we will perform the similarity computation substantially faster (spending time less than $d$). The second algorithm will keep the dependence on $d$, but will reduce the dependence on $n$ — instead of scanning all the sets to detect a similar object, we will do something somewhat smarter to zoom in directly on the sets which are close.

Both algorithms will provide substantial savings, and are widely used in practice. However, this improvement is not free. The algorithms we will see will not be exact; they will use both approximation and randomization. So we will relax our guarantees on two fronts:

- The algorithms will only work with some probability.
- We will not be able to detect exactly the condition $J(A, A_i) \geq s$; instead, we will be able to detect something more approximate.

## §22.3 Approximate Similarity Search

We'll now set up what we mean by an approximation.

**Problem 22.6.**

- GIVEN — $n$ sets $A_1, \ldots, A_n$ (which are subsets of $U$), and *two* similarity thresholds $s$ and $s'$ in $[0, 1]$, with $s > s'$.
- GOAL — to build a data structure that, given a query set $A$:
  - As before, if $J(A, A_i) \geq s$ for some set $A_i$, our algorithm returns YES.
  - If $J(A, A_i) < s'$ for all sets $A_i$, our algorithm returns NO.

Since $s > s'$, the two cases are disjoint, so the algorithm is never in a situation where it has to report both YES and NO. However, as we can see, the specification does not cover all the cases — it is perfectly plausible that there is no set with similarity at least $s$, but there may be some set with similarity slightly less than $s$ (and more than $s'$). In that case, the algorithm is unspecified — it can do whatever it wants (it can report YES or NO or refuse to give an answer). This is sometimes called a 'promise problem' (where we are promised that only one of these two cases happen).

Note that if $s' = s$, then the problem would be the exact same as before — then the two cases would be exhaustive (either there is a very similar set, or all sets are not very similar).

Here there is a gap between YES and NO cases, which gives us space for approximation; our algorithms will exploit this gap to find faster solutions.

**Student Question.** *What do we do in the gap in practice?*

**Answer.** The algorithms that we will see essentially work in the following way — if there is a set with similarity at least $s$, they report some (potentially different) set with similarity at least $s'$. This will hopefully become more clear after we see the algorithms.

**Student Question.** *Do we choose both $s$ and $s'$, or does the algorithm choose $s'$?*

**Answer.** Both $s$ and $s'$ are given (the algorithm doesn't choose them — we have a collection of sets, and the user sets the similarity threshold).

**Student Question.** *Why don't we set $s = s'$ then?*

**Answer.** If $s$ is separated from $s'$, this will make it easier to come up with a good solution.

## §22.4 Algorithm 1

Our first algorithm will reduce the dependence on $d$ (but still perform a linear scan). On a high level, the basic idea is to speed up the computation of

$$J(A, A_i) = \frac{|A \cap A_i|}{|A \cup A_i|}$$

by performing *random sampling*. The probability that a uniform random element from the union falls in the intersection is exactly the similarity measure that we want to estimate (since it's defined as the size of the intersection over the size of the union).

What this means is that instead of computing the intersection, we can instead select $t$ samples $a_1, \ldots, a_t$ from $A \cup A_i$, and check for each element whether it belongs to both set. If $s$ samples fall into both sets, then

$$\frac{\mathbb{E}[s]}{t} = \frac{|A \cap A_i|}{|A \cup A_i|}.$$

More strongly, by tail inequalities such as the Chernoff bound, we actually know that

$$\frac{S}{t} = \frac{|A \cap A_i|}{|A \cup A_i|} \pm \varepsilon$$

with high probability.

This gives us a relatively fast algorithm, as long as we manage to figure out how to sample from the union of two sets without actually reading all of them.

**Student Question.** *Can't we sample from the union by sampling from both $A$ and $A_i$ and dividing the result by 2?*

**Answer.** This doesn't quite work because for example, if the two sets are equal, then you're basically sampling from the same set, computing a similarity of 1, and dividing the result by 2; this tells you that two equal sets have similarity $1/2$.

### §22.4.1 Random Sampling

The idea of how we'll sample uniformly at random from $A \cup A_i$ is simple, but a very neat trick and remarkably useful.

In principle, it might not be possible to do this. But this is a data structure problem, which means we can do some preprocessing; and this preprocessing will be what we need to do this.

The first thing we do is define a *rank* function $h: U \to [0, 1]$, selected uniformly at random — so for every element in the universe, we assign it a number selected uniformly at random (which we call its *rank*). Then we can extend the notion of ranks to arbitrary sets, by defining

$$h(A) = \min_{a \in A} h(a).$$

**Example 22.7**

Suppose that $A = \{a, c, d\}$, and we map $a \mapsto 0.5$, $c \mapsto 0.1$, and $d \mapsto 0.2$. Then $h(A) = 0.1$.

So each set is represented as the minimum rank of any element in the set.

**Fact 22.8 —** For any sets $A$ and $B$, we have

$$\mathbb{P}(h(A) = h(B)) = J(A, B).$$

In other words, by representing each set by its minimum rank, from this alone we can represent the Jaccard coefficient.

*Proof.* Each element in $A \cup B$ is equally likely to be assigned the minimal rank in $A \cup B$ (the ranks are random, so every element has equal chance of being the smallest element). Suppose this element with minimal rank is $m$. Then $h(A) = h(B)$ if and only if $m$ is in $A \cap B$ — if $m$ is in $A \cap B$ then of course it's both the minimum of $A$ and $B$, while on the other hand if $m$ is in $A$ but not $B$, then the minimum rank of $A$ will be smaller than the minimum rank of $B$.

So then $\mathbb{P}(h(A) = h(B))$ is the probability that the element $m$, which was selected uniformly at random from $A \cup B$, is in $A \cap B$. And the probability that a random element from the union belongs to the intersection is exactly $J(A, B)$. □

**Student Question.** *If we are fixing $h$ at the beginning, how does this work with multiple sets?*

**Answer.** In general the events $h(A) = h(A_1)$, $h(A) = h(A_2)$, and so on will not be independent. But we won't need them to be — the result for two sets is enough.

**Student Question.** *What happens if two elements have the same rank?*

**Answer.** In the theoretical perspective, two elements collide with probability 0 (since they're real). In practice we would have to use floats, but the probability they collide is still tiny, and can be ignored.

**Student Question.** *Are all elements in the set assigned a unique rank?*

**Answer.** Yes — all elements in the whole universe get assigned a rank, so in particular all elements in each set is assigned a rank. These ranks are all unique, since they're real numbers.

**Student Question.** *Why do we use* min *and not* max*?*

**Answer.** They are symmetric, so either would work. But either choice is natural — we need to be able to summarize a set by one of its ranks, and the minimum happens to work (i.e. it ensures the property we want).

So what we've done is that we're now able to summarize every set into a single number, such that the probability any two sets are equal is exactly the quantity we want to estimate.

**Algorithm 22.9 —** During preprocessing, we select $t$ rank functions $h_1, \ldots, h_t$ (where each assigns a random rank to every element of the universe). Then for every set $A_i$, we compute the vector of ranks

$$g(A_i) = [h_1(A_i), h_2(A_i), \ldots, h_t(A_i)].$$

Now each set $A_i$ is represented by $t$ ranks.

Then given a query $A$, we first compute

$$g(A) = [h_1(A), \ldots, h_t(A)].$$

For each $i$, we then enumerate all the sets, but instead of directly computing the similarity, we estimate it by looking at the two vectors $g(A)$ and $g(A_i)$, and calculating the fraction of positions at which they agree — we define

$$\text{Sim}_i = \text{Sim}(g(A), g(A_i)),$$

where $\text{Sim}(x, y)$ is the number of positions with $x_i = y_i$. Then if $\text{Sim}_i \geq s - \varepsilon$ for some $i$, we answer YES; otherwise we answer NO.

**Student Question.** *How do we generate these rank functions without storing tables of size $U$?*

**Answer.** In practice, we are not generating a completely random number for each word (since the number of words is in the billions). Instead, we use a pseudorandom generator (with a seed). One way of thinking of this is that earlier in the course we talked about universal hash functions, which let us avoid creating a random number for each element; it's a bit more complicated here, which is why Prof. Indyk tried to sweep it under the rug, but it is important.

**Student Question.** *Is there $s$ and $s'$ in this case?*

**Answer.** Here $s$ is just $s$, while $s'$ will depend on $s$ and $\varepsilon$.

**Question 22.10.** How much time does this take?

First, when we perform pre-processing, we run through all $n$ sets, take time $d$ to compute the minimum rank of each set, and we do this $t$ times; this takes $O(ndt)$ time. But this is done in preprocessing; that's done at the very beginning and only once, so we can afford to spend more time.

Then when we answer a query, we make a pass over all $n$ sets. But now to estimate the similarity, we only need to make a pass over the vectors which are of size $t$, not size $d$. So the saving comes from replacing our sets with an approximation by a smaller vector, which saves an order of magnitude of time — in practice $t = 100$ often suffices.

This gives us runtime of $O(nt)$.

**Student Question.** *Are we assuming $n \geq d$ implicitly?*

**Answer.** Yes, since calculating $g(A)$ takes $dt$ time. In practice this is true ($n$ may be in the billions, and $d$ in the thousands).

*Proof of Correctness.* By the Chernoff bound, we can show that if we sample $t \asymp \log(n)/\varepsilon^2$, then $\text{Sim}_i = J(A, A_i) \pm \varepsilon$ for all $i$ with probability at least $1/2$. (The reason for $\log(n)$ is because we are taking the union bound over $n$ elements — this means we need a probability of $1/2n$ for each $i$.)

Now if $J(A, A_i) \geq s$, then $\text{Sim}_i \geq s - \varepsilon$, so our algorithm says YES. On the other hand, if $J(A, A_i) < s - 2\varepsilon$ for all $i$, then $\text{Sim}_i < s - \varepsilon$ for all $i$ as well, so the algorithm says NO. (Here $s' = s - \varepsilon$.)

So the algorithm correctly solves the promise problem as long as our event occurs, and it occurs with probability at least $1/2$. □

# §23 December 6, 2022

## §23.1 Introduction

In the first lecture, we saw a bird's eye view of what the class is about. It can be summarized as a lot of techniques — some old (like divide and conquer) and some new (like linear programming and approximation algorithms). We've learned a lot of techniques that can help us solve algorithmic problems.

But there's another (smaller) part of the class, about *computing models*. Most of our algorithms operate in the RAM model — where we can read from memory, write from memory, perform operations, and so on. But there are other models.

In particular, one different model of computation is *online algorithms*, which we've seen already — an online algorithm is when we only get one piece of data at a time, and we can't see into the future — the input is not written in advance, but instead comes to us one by one, and we have to make decisions in real time which we can't revoke. This is more restrictive — we're losing something because we can't see everything. We can measure the quality of such an algorithm by comparing ourselves with the optimal offline algorithm, for instance.

This is particularly useful in applications such as networking — where an algorithm needs to receive and send packets over a network (once you send a packet you can't unsend it, and at no point do you see the full network — you can only see a particular local view).

Today we're going to see another model, called *streaming algorithms*. They'll share some similarities with online algorithms, but there are fairly different objectives and ideas.

Today we will focus on algorithms that are *sublinear* (i.e. $o(n)$) — we'll mostly focus on algorithms that use sublinear *space*, but we'll also see some that use sublinear *time*.

## §23.2 Sublinear Space Data Stream Model

**Definition 23.1.** Iin the data stream model, the input is a bunch of items $i_1, i_2, \ldots, i_n$ coming from some universe $U$. (The length of the input $n$ may or may not be known in advance.)

The algorithm can only perform one pass over the input (from left to right).

Furthermore, the algorithm has **limited storage** — an amount of storage that is $o(n)$.

In all examples we'll see today, we'll actually only have $O(1)$ storage. (The last condition is needed because otherwise we wouldn't be requiring much — we could just read the input and copy it somewhere else.)

**Example 23.2**

If $U = \{0, \ldots, 9\}$, then one possible input is 8, 2, 1, 9, 1, 9, 2, .... The algorithm then reads these elements from beginning to end, and at the end it's supposed to record the answer it's designed to give.

We'll now see a bunch of examples of algorithms that fall into this model. But first, for some motivation, where might such algorithms be useful? If we have terabytes of information stored in a disk or tape, the only thing we can reasonably do is make a pass over the data and perform all our operations on the fly (we don't have enough memory to put everything into memory and have random access). On a disk, accessing information stored there is much more efficient if we start from the beginning and scan to the end. (It would take a long time to access a random element from the disk.) And of course, we can only use as much memory as our computer has (which may not be constant, but has to be much smaller than the input).

## §23.3 Simple Examples

**Problem 23.3.** Compute the minimum of the elements $i_1, i_2, \ldots, i_n$.

We can do this by first assigning $i_1$ as the minimum, and then going through the stream and comparing each new element with the current minimum — the main point is that at every given time, we only need to know what the minimum is so far. So we can maintain `current_min` initialized to $i_1$, and every time we see a new element, we set

$$\texttt{current\_min} = \min(\texttt{current\_min}, i_j).$$

It's clear this only reads from left to right, and only keeps track of one element in space.

**Problem 23.4.** Compute the *sum* of the elements $i_1, i_2, \ldots, i_n$.

This can be done in essentially the same way — we keep track of the running total sum, and every time we see a new element, we just add it.

## §23.4 Majority Element

Now we'll solve something which is not hard, but is also not that easy.

**Problem 23.5.** Given a stream of elements, find one element that occurs more than $n/2$ times (if one such element exists).

If $U$ is small (e.g. all digits), then we can allocate a counter table (with one counter for each digit) that keeps track of the number of times each digit occurs. This would take $U$ space. This is useful if our universe is very small (for example, if we're counting votes). But in many applications, the universe may be very large, and then we can't afford to keep $U$ counters.

We could attempt to do this in general by hashing — by hashing an element into a table the first time we see it, and incrementing its count afterwards. But the problem is that now we'd be using $n$ space.

**Student Question.** *If there's no majority element, do we have to return that, or can we do anything?*

**Answer.** In this formulation, the algorithm can do anything when there's no majority element. (This formulation is more of a toy problem, but some natural generalizations we'll get to soon are actually used very widely for data processing.)

The algorithm that does the job is the following:

```
counter = 0
majority_element = None
for each item i_j:
    if majority_element = i_j then counter = counter + 1
    else if counter = 0 then
        majority_element = i_j
        counter = 1
    else counter = counter - 1
return majority_element
```

The algorithm maintains two variables — our candidate majority element (which will switch over time as our algorithm sees more and more data), and a *counter* which intuitively is supposed to count the number of times the candidate occurs (although this is not exactly true).

For each element we see, there are two cases. If the new element is equal to the current candidate, then the counter increases. Otherwise, if the new element isn't the candidate, then we have two cases. In the first case, if the counter of the candidate is 0, then we replace the candidate with the new element and set the counter to 1 (the running count of the new element). Finally, if we see an element which is *not* the majority element, but the current counter is greater than 0, then we decrement the counter.

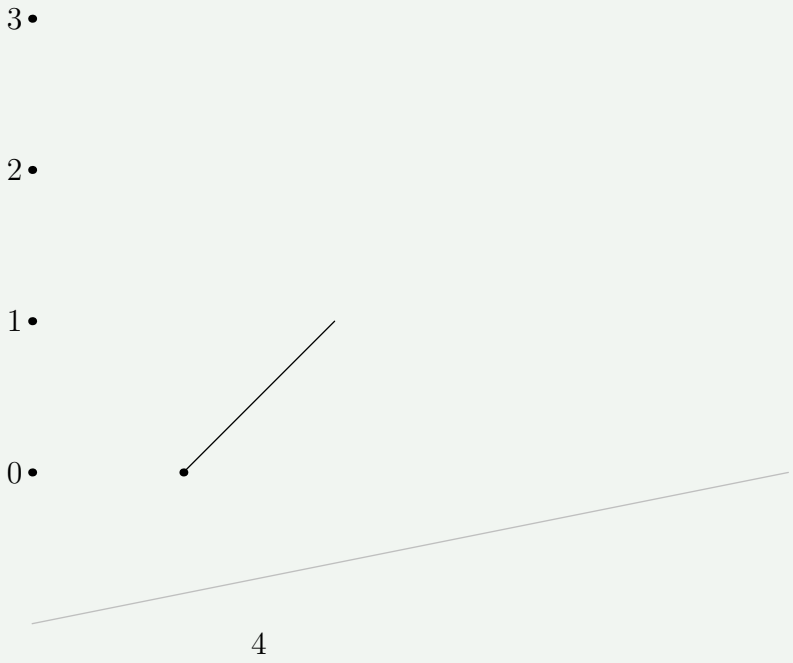
It's quite non-obvious to see what this algorithm is doing, so we'll see an example.

### Example 23.6

Suppose we have the stream 4000102032002; then 0 is the majority element, since it occurs 7 times (out of 13). So the algorithm has to do something, namely to report 0.

We'll visualize the counter on the $y$-axis, and write down the candidate below on the $x$-axis. The counter starts at 0:


After the first step, the current candidate becomes 4, and the counter becomes 1.




The next step, we read $0 \neq 4$. The counter is 1, so we keep the candidate 4 but decrement the counter to 0.

So throughout the suffix of our stream, we maintain the correct candidate and we return it in the end.

> **Student Question.** *If the second number was* 5 *instead, wouldn't the algorithm return the same thing?*
>
> **Answer.** Yes, but that's fine — we no longer have a majority element, so the algorithm is free to do whatever it wants. The funny thing about the formulation is the algorithm is supposed to do something useful if there's something useful to report, but it won't tell you whether that's actually the case. (Sometimes you can make another pass to actually count the number of 0's.)

We still need to show that the algorithm will actually output the majority element.

*Proof.* First assume the majority element exists (if it doesn't, we are done); for the purpose of reasoning, let's assume the majority element is 0.

We can split the timeline into intervals starting and ending when the counter reaches 0.

The key claim is that in each interval except possibly the last, the fraction of 0's is at most $\frac{1}{2}$.

So when we arrive at the last interval, 0 must still be the majority element. And this means when we pick a candidate here, this candidate must be 0 (since if it were not, the 0's seen in the interval would decrement the counter to be below the 0-level).

> **Student Question.** *How do we know that at each interval except the last, the fraction is at most* $\frac{1}{2}$*?*
>
> **Answer.** There are two cases. One is where the majority candidate at the beginning of that interval is not 0, say 4. Then every time we see 4 we incremenent, and every time we see not-4 we decrement. We start and end at 0, so exactly half of the elements we see are 4's; this means at most half of the elements can be 0's (since half are not-4's).
>
> The other case is when the candidate is 0. Then we increment when we see 0 and decrement when we see something nonzero. So in this case, we'll see *exactly* half zeros (since we increment and decrement an equal number of times).

□

### §23.4.1 Generalizations

The problem we solved is somewhat a toy problem — it finds the majority if it exists, but it doesn't even tell us whether we've found something useful or not.

Both of these issues were addressed in a really elegant paper by Misra–Gries 1982. They proposed a generalization of this argument that uses $k+1$ counters to find all elements that occur more than $n/k$ times, using $O(k)$ space. (Previously we had $k = 2$.) Such elements are often called 'heavy hitters.' This formulation is particularly useful — if you want to detect a denial of service attack in networking, you can count the number of times a certain source is dominating.

The algorithm can even estimate the number of times those elements occur, up to an additive error of $O(n/k)$. (So if $k = 100$, we can get the number up to 1% of $n$.) This is the useful version of the algorithm.

Note that this algorithm has an error of $O(n/k)$, so if there are elements just at the threshold, we may not be able to verify whether they occur exactly 1% of the time or not. But our error will only be 1%. ($k$ is a parameter; in practice the fraction is often set to 0.1%, which means $k = 1000$; if we're processing a dataset of size a billion, this is a huge win.)

## §23.5 Random Sampling

The next streaming algorithm we'll cover is an algorithm for an equally natural and simple problem:

**Question 23.7.** How do we randomly sample from the stream?

**Problem 23.8.** Given a stream $i_1, \ldots, i_n$, we want to report an element $r$ chosen uniformly from the stream.

If we know $n$ in advance, then the problem is trivial — we generate a random index $j$ from $[n]$, do nothing until $i_j$ arrives, store it, and report it.

But for that to work, we need to know the length of the stream. So what happens if we *don't* know $n$ in advance?

This can actually be done in $O(1)$ space:

**Algorithm 23.9** (Reservoir Sampling) — We keep a current candidate for the random element, denoted by $r$. Whenever we see a new element $i_j$, we set $r$ to this element with probability $\frac{1}{j}$, and keep $r$ unchanged otherwise.

This is very simple and very useful — it allows us to get a random sample from a lot of data, with very low space overhead.

**Example 23.10**

If we have only one element, of course we keep it.

If we have two elements, then on the secnod step $i_2$ replaces $i_1$ with probability $1/2$.

*Proof.* We'll prove correctness using induction. Suppose that after step $j$, $r$ is chosen uniformly at random from $i_1, i_2, \ldots, i_j$ (i.e. $\mathbb{P}(r = i_j) = \frac{1}{j}$ for all $t \in [j]$). Then we want to show that this extends to the next step.

After step $j+1$, we have

$$\mathbb{P}(r = i_{j+1}) = \frac{1}{j+1}$$

by design (since we're keeping $i_{j+1}$ with this probability). Meanwhile, for the other elements $i_t$ (with $1 \le t \le j$), we have

$$\mathbb{P}(r = i_t) = \frac{1}{j} \cdot \frac{j}{j+1} = \frac{1}{j+1}$$

(since it had to have been the candidate after the previous step — which has probability $\frac{1}{j}$ by the inductive assumption — and we had to have not replaced it). □

## §23.6 Approximate Diameter of a Set in a Metric Space

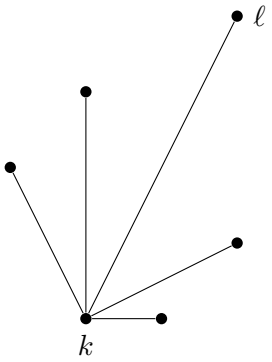
We'll now see an algorithm which runs in sublinear *time*, not just space (of course any algorithm which is sublinear in time is also sublinear in space, since it doesn't have enough time to use more space — but the converse is not true, and in general sublinear time algorithms are much more rare).

**Question 23.11.** Suppose we have a distance matrix $D$ between a set of $m$ points, where $D_{ij}$ is the distance from $i$ to $j$ — and these distances satisfy the triangle inequality and symmetry (i.e. $D_{ij} = D_{ji}$ — going from $i$ to $j$ is the same as going from $j$ to $i$ — and $D_{ij} + D_{jk} \geq D_{ik}$ — going from $i$ to $k$ by stopping at $j$ cannot make the distance shorter).

We want to approximate the *diameter* of our set of points — i.e., $\max D_{ij}$. Our goal is to approximate it by a factor of 2 — if $i$ and $j$ are points that maximize the diameter, we want to find two points $k$ and $\ell$ such that $D_{k\ell} \geq D_{ij}/2$.

Note that the input has size $n = m^2$. Our algorithm is going to be sublinear in the size of the matrix, not in the number of points. (We could look at all pairs of points in the matrix and take the max to get an exact answer, but that would take linear time.)

**Algorithm 23.12 —** Choose any *one* point $k$ (in any way), and compute the distances from this point to every other one. Pick $\ell$ to be the point with the largest such distance.




It's clear this takes $O(m) = o(n)$ time, since we simply enumerate the distance to all other points.

*Proof.* We know by the triangle inequaltiy that

$$D_{ij} \leq D_{ik} + D_{jk},$$

where $k$ is our anchor point and $D_{ij}$ the longest distance. But we know that both of these distances are at most the distance from $k$ to $\ell$, since we chose $\ell$ as the point furthest away from $k$ (and by symmetry). So we then have

$$D_{ij} \leq D_{ik} + D_{jk} \leq D_{k\ell} + D_{k\ell} = 2D_{k\ell},$$

and we are done. □

**Student Question.** *If we iterate this more times, can we get a better approximation?*

**Answer.** Interestingly, the answer is no — we can get more accurate algorithms for particular types of metrics (such as a tree), but if the only thing we know is that the matrix satisfies the triangle inequality and is symmetric, then a factor of 2 is the best we can do in sublinear time. (You can imagine the case where the only way to find the true value is by scanning the whole matrix.)

# §24 December 8, 2022 — Distributed Algorithms

Today we'll also cover a computational model — *distributed* and *parallel* algorithms. We won't see the full model because it's quite complicated, but the simple model we'll use will still give lots of insight. With this model we'll solve two problems — leader election (it's hard to do anything in computers without a leader; we'll show when it's possible and not possible) and computing the *maximal* independent set. This algorithm is a real gem; hopefully at the end of the lecture we will appreciate the beauty as much as Prof. Indyk does, but even if we appreciate it half as much, that will be a success.

## §24.1 Distributed Algorithms

These days, whether we like it or not, performing computation in a distributed or parallel model is basically the way to go. The internet is a bunch of independent computers loosely connected somehow — it's one giant computational tool, but with many processors or entities participating in it. This is maybe the biggest distributed system on the planet, but there are others. For example, multiprocessors — a computer has multiple cores, and whenever we have more than one entity doing something, coordinating who's doing what becomes a key issue.

Once we get to more than one processor doing things, suddenly we have all sorts of issues from things happening at the same time. There are problems if things happen at exactly the same time, since two actions may contradict each other and there's no time to decide. Things can also go wrong if there's an uncertainty of timing — because if we perform an action that relies on something that should've happened before, then if we don't know when that's happened we don't know whether to do our action.

There's also things like failures — some processors may stop working (on the internet, that's fairly typical).

So when we have parallel or distributed ways of doing things (with two processors doing something), we have to deal with various issues. (If we have ever done a process with a group size of more than 1, we may have dealt with these issues — where our partner was supposed to do something but didn't in time, or when the partner suddenly disappeared. In the real world there are also worse situations — if your processor can start behaving adversarially (which is uncommon in group projects but can happen on the internet, where you have adversaries who may want what you're doing to crash).)

But the good news (or bad news) is that today we will try to abstract away from most of these problems, and focus on a relatively simple model. It definitely doesn't capture everything, but it's still very useful to see what can be done and what are the issues when we do things at the same time.

## §24.2 Synchronous Network Model

**Definition 24.1.** In the **synchronous network model**, our system is modelled as a directed graph. At each node of the graph is a processor. The processor can have some *local* memory (with parts of the input). There is no shared memory, so if we want some other processor to know something, we have to communicate it.

Communication happens along the edges — out-edges send messages out, and in-edges let us receive messages from other nodes. We assume every node knows its out-neighbors and in-neighbors (and can distinguish between where messages are coming from), but that's all they know — their local input and their neighborhood, and all the messages that have been exchanged.

We assume that there is a common clock, and all messages and computation happen according to that clock — in step 1 messages are sent out and in and there is some processing, and then step 1 ends and everyone knows this. Then after the first step is finished, there is a second step where everyone exchanges information and computation is performed, and so on.

This is in some sense an idealized Ford factory model — everyone does something at particular times, and there are no processor delays or failures. There is also no adversarial behavior — in this model we assume that it is not possible for a node to start acting not according to the algorithm.

This is the model we will use in today's lecture.

> **Student Question.** *What do we know about this digraph?*
>
> **Answer.** We will focus on one particular type of network. In general, each node might not know the network fully — but for the algorithms we are going to see, we will consider a particular type of network where all nodes know what the network is.

(For the problems we will consider, there will be almost no input. The graph itself will be the input, and the system will want to execute certain tasks defined based on the network.)

The key fact is that knowledge is *local* — some process can easily figure out the information at its neighbors. But if the diameter of the network is big, it might take a while before the endpoints can even interact with each other.

> **Student Question.** *When a node sends a message, does the recipient receive the message in the same round or the next?*
>
> **Answer.** The same round, though it doesn't really matter (since we could put a constant number of rounds in one).

The 'constant clock' means that there is a well-defined notion of the beginning and end of each step — there is no misunderstanding of when the step started or ended. So for example, we can't have some processor get into some time-warp where it's still executing step 1 while everyone else executes step 2 or 3.

> **Student Question.** *Can you only send a message across one edge per round?*
>
> **Answer.** Yes. When we're calculating the cost of our algorithm, we will care about the total number of steps (parallel time) and the total number of messages. We will not care about the internal computation.

## §24.3 Leader Election

> **Problem 24.2.** The goal is to distinguish *exactly one* node as the leader — the leader has to output 'I am the leader' while no other node outputs anything.

There is a variation of this problem where one node says 'I am the leader' and every other says 'I am not the leader.' This can also be done (with a similar algorithm), but it takes a bit more work; so we will just be happy with one node declaring itself a leader, and no other node declaring itself a leader.

This declaration is really in the head of the processor; once it does this, it can message its neighbors and so on. Of course if one node at the left end of the network declares itself a leader, nodes on the other side of the network might not know who the leader is. When our algorithm ends, each node should have either declared itself a leader or not declared itself a leader; this means the node knows whether it is or is not the leader (but does not necessarily know who the leader is).

Why solve this problem? Whenever we have any group that wants to do something, things become complicated without a leader (if everyone's a leader then no one is, and coordination becoems difficult — at the least this is certainly true for processors, where anarchy may not be the way to go if you want your processors to do something quickly).

This can be done in general, but we will focus on one of the simplest possible graph topologies — a ring network with $n$ nodes connected in a cycle, directed in both directions. We will solve the leader election problem for this particular network (even here the task is quite nontrivial).

### §24.3.1 Impossibility

It's crucial what the nodes know or have. For our purposes, we assume that the nodes just have a blank slate — they don't have names or IDs or any other way to distinguish themselves. They just know their neighbors as 'clockwise' and 'counterclockwise' but they don't know who they are, so to speak.

**Theorem 24.3**

If all processes are deterministic and identical, then it is impossible to elect a leader.

(This is not a statement saying it's NP-hard or computationally hard; it's just not possible at all.)

This is perhaps an interesting philosophical statement — if there's no way to distinguish anything, then there are obvious tasks you can't accomplish. The proof is almost boring because it follows the obvious intuition — if everyone's exactly the same, there's no way for the nodes to distinguish themselves, and after every step everything will have to remain the same, and an election of the leader would break the symmetry (which isn't possible).

*Proof.* Assume that we have an algorithm that solves the problem. First we should be careful about what the processor knows — we can assume it has a *communication history*, a *program counter* (the code being used), and some *local memory*. We call these together the *state* of the processor.

At the very beginning, all processors start in the same state. But then by induction, this must remain true — if we assume after $r$ rounds that all processors are in the same state (i.e. they generate and receive the same messages, they follow the program code identically, and have the same memory content), then they will still send and receive the same messages and perform the same computation, so they will remain identical.

But at the same time, we assumed the algorithm solves the problem, meaning that someone gets elected. But given that the states are all exactly the same, this means everyone gets elected; this is a contradiction. □

In some sense there is nothing deep here; it is mostly just saying that if things are identical at the beginning of a step, then they are still identical at the end. Everything else is just writing what exactly the code is and what the processor knows.

**Student Question.** *Does this only hold for a ring network?*

**Answer.** Yes. If different nodes have different numbers of neighbors, then the symmetry may be broken. But you can imagine other symmetric networks, where it still works; if the network is symmetric and we start from a symmetric state, then we will persist in one.

### §24.3.2 The Importance of Identity

To overcome this theorem, we have to break the assumption. To do so, we will assume that all nodes have a unique identifier (UID) — i.e., everyone has a name, and their names are all different.

First, how can we ensure this? There are multiple ways of doing this. One is to allow randomized algorithms, where every processor can generate a random real number in $[0, 1]$ and set this number as its identity. Another option is that the processors may have serial numbers, which are unique.

We will see that as long as the processors have unique IDs, we can use them for leader election.

**Algorithm 24.4 —** We will choose the processor with the maximum valued UID to be the leader. To communicate which node has the maximum valued UID, we do the following: at every step, every node sends the value of the biggest UID it has seen so far (including its own) to the right. When a node receives its own value from its neighbor, then it declares itself the leader.

So at the very beginning, each node sends its own ID to the right. It also receives some ID and computes the maximum of its own and the one it received. The next step it will send the updated maximum (of its own ID and the received one). It keeps updating the maximum, and life goes on; but if at some point one process receives its own ID coming back, then this node declares itself a leader.

The intuition is that if a node receives its own ID, then its own ID went all the way around the network and came back. So it must be the maximum, because otherwise if there was something larger, then it'd have been updated.

At this point the node knows it's the leader. Other nodes may not know what's going on, but they certainly do not declare themselves the leader.

Interestingly, this works even if the nodes don't know $n$. (If they know $n$, then they can all run the process for $n$ steps, and if they didn't receive the maximum then they know they're not leaders. But even if $n$ is not known, they can continue performing this process — except the node which is not a leader, who knows that the task is finished.)

**Theorem 24.5**

If $i_{\max}$ is the process with the maximum UID, then $i_{\max}$ outputs 'leader' at the end of round $n$, and no other node outputs anything.

The proof is conceptually simple, but it is remarkably tedious (it carefully goes over who knows what and when), so we won't go over it. (This is a feature of distributed algorithms — many things are intuitive, but proving them formally can be quite hard. The flip side is that many things are intuitively simple but are wrong — decades ago there were famous algorithms published and peer-reviewed, which were out there for several years until someone realized they were incorrect. It's remarkably difficult to reason about distributed algorithms. The worst things happen in asynchronous models where some things may or may not have been sent — where some message that intuitively comes first actually may not. Subtle questions like this will not be on the exam.)

To finish this, we can analyze complexity. There is two notions of effort. One is the number of rounds until someone declares themselves a leader; this is $n$ (which follows directly from the theorem). Another measure is the communication — the total number of messages sent (which is important because every message takes some amount of energy). Here the total amount of messages sent is at most $n^2$ — there are $n$ rounds, and in each round each of the $n$ nodes sends a single message.

This is not a very energy-efficient process; there do exist more message-efficient algorithms that run in $O(n \log n)$ (which we are not going to see).

**Student Question.** *Do we also care how much memory is used by a node?*

**Answer.** Usually no. But in reality we cannot use infinite memory. The good news is that in this particular setting, maintaining the maximum is very memory-efficient — similar to streaming algorithms.

**Student Question.** *Is there an algorithm for leader election in more general graphs?*

**Answer.** Yes; for a graph with diameter $d$, it can be done in $\Theta(d)$ steps.

## §24.4 Maximal Independent Set

The next question we'll consider is computing a maximal independent set. Crucially, we want a *maximal* independent set and not a *maximum* independent set.

**Problem 24.6.** Our input is a graph. We want to find a maximal independent set. Our goal is to find an algorithm that guarantees at the end, every node knows whether they are in the independent set or not.

**Definition 24.7.** In an undirect graph, a subset of nodes is **independent** if no two nodes in the set are neighbors.

**Definition 24.8.** An independent set is called **maximal** if we cannot add any more nodes to it while preserving the fact that it is an independent set.

Note that maximal doesn't mean *maximum* — e.g., in a 6-vertex star, taking only the center node gives a maximal independent set, but it's obviously not the maximum (we could instead take every spoke to get an independent set of size 5). In particular, finding a maximum independent set is NP-hard, while finding a maximal independent set can be solved in $O(V + E)$ time in the sequential setting (we take any node, put it in the set and remove its neighbors, and so on).

Here we are instead in the distributed setting — we have a network of nodes, which are allowed to generate random numbers and operate in parallel and synchronously. After some number of rounds, we want each node to declare itself in the set or not (and the output should be a maximal independent set).

**Student Question.** *Do they have to output whether they are in the set simultaneously?*

**Answer.** No — they can declare themselves to be in or not in the set at different times, but at the end everyone should be declared.

The algorithm will be randomized.

**Student Question.** *How is this different from the sequential problem?*

**Answer.** Here the network is the input — in particular, one node has some information about its neighbors, but it has no clue what the rest of the network looks like. (In everything we've seen so far we had one central processor that knew all the input and could perform the computation; here that is not true.) Also, the computation is done in parallel — everyone does something in the first round, then there's a second round, and so on. So there's no single central authority doing the job.

*Do the processors know $n$, so that they know how many steps they have to take?*

***Answer.*** *We can assume that. It's not actually crucial for executing the algorithm, but we can assume it for now — the algorithm will actually not use $n$ (only our analysis for how long something will run will use $n$).*

One 'ideal' solution that's good for intuition is to essentially simulate the sequential algorithm. The sequential algorithm picks a node, puts it in the independent set, and removes all its neighbors (and then iterates). To implement this here, we could pick one node by leader election (this isn't the exact same thing — some nodes are already in or out and don't participate — but it's essentially the same). Then we set the leader to YES and its neighbors to NO, and we continue until all vertices are labelled.

This isn't so much a full algorithm as a thought experiment to see how to do it. Unfortunately, if we do this the total number of rounds will be very large, because we're trying to simulate a sequential process in parallel — each leader election takes $O(D)$ rounds where $D$ is the diameter, and so the runtime is $O(nD)$.

We'll see how to improve this to $O(\log n)$ rounds! Here we won't simulate the sequential algorithm, and instead we'll actually make decisions independently, in a way that still ensures the result is correct.

**Algorithm 24.9** (Luby's MIS algorithm (randomized)) — At the start, all processors are *active*.

In each round, each active node $i$ performs two steps.

In step 1, it chooses a random value $r_i$ from $[0, 1]$ and sends it to its neighbors, and it receives values from its neighbors. If all the received values are less than $r_i$, then it *joins* the MIS (i.e. it declares itself to be in the MIS).

In step 2, if a vertex joins the MIS, then it announces this to all its neighbors. If it has received such an announcement, then it decides *not to join* the MIS (i.e., it declares that it is not in the MIS).

If the node decided either to join or to not join in this step, then it becomes *inactive*.

The idea is that in each step, independently from other rounds, each node generates a random number and receives those of its neighbors. If locally its number is the largest, then it joins the maximal independent set.

The idea is kind of similar to leader election — the maximal wins — but it's important that here, it's done locally (it *locally* has the largest value; it doesn't wait to find out whether it's *globally* the largest).

We are going to argue first that the process is correct (once all nodes have made a decision and become inactive, the result is really a maximal independent set), and that all nodes become inactive after $O(\log n)$ rounds with high probability.

To prove correctness, there's two things we need to show:

**Lemma 24.10**

The final set is independent.

*Proof.* A node $i$ joins the set if it is locally maximal; this means none of its neighbors are locally maximal, so we cannot include two neighbors in the same step (and the neighbors are then eliminated). □

**Lemma 24.11**

The final set is maximal — i.e., we cannot add any more nodes.

*Proof.* A node only becomes inactive when it itself or one of its neighbors decides to join the MIS. If the node joins then of course we can't add it again; and the only way the node declares that it's not in the set is if one of its neighbors is added in (in which case we can't add this node). So we can't have a situation with any node not in the set with no neighbors in the set. □

Now let's analyze runtime. Here we are only going to analyze it for the ring network; the same holds for any network with constant degree, but it's easier to reason about it in this case.

**Lemma 24.12**

The number of rounds for teh algorithm to terminate is at most $2 \log_2 n$ with probability $1 - 1/n$.

*Proof.* We define an edge $(i, j)$ to be *active* if both its nodes are active.

**Claim 24.13 —** For every edge $(i, j)$ and every round it starts as active, it will become inactive with probability at least $1/2$.

(We will show this soon; first we'll see how this gives us the desired result.)

Then by the union bound, the probability any edge remains active after $\ell$ rounds is at most $1/2^\ell$. Setting $\ell = 2\log_2 n$ and using the union bound gives the desired result (as each edge is still active with probability at most $1/n^2$, and there are $n$ edges).

Now we'll prove the claim. Suppose our edge is $(i, j)$.

**Case 1** (Both edges incident to $(i, j)$ are inactive). That means the two nodes next to $i$ and $j$ are already inactive. In this case, either $r_i < r_j$ or $r_j < r_i$. These two nodes only care about each other (since their other neighbor is asleep), so one of them will join the MIS, and the edge will definitely become inactive.

**Case 2** (There is one active edge incident to $(i, j)$). Without loss of generality suppose that neighboring edge is on the side of $j$. Then with probability $1/2$ we have $r_i > r_j$. Since the other neighbor of $i$ has already gone to sleep, if this occurs then $i$ joins the independent set, and the edge becomes inactive.

**Case 3** (Both edges incident to $(i, j)$ are active). Suppose we have the path $(t, i, j, w)$. Then something similar still happens — first, $r_i$ is the largest among its neighbors with probability $1/3$ (every node has the same chance of being the maximum). By symmetry, the same is true for $j$. These two events are disjoint (either $r_i > r_j$ or vice versa), which means the probability that either $r_i$ or $r_j$ is a local maximum is $2/3 > 1/2$.

So in each of these situations, with probability at least $1/2$, one of the endpoints of the active edge becomes inactive. □

**Remark 24.14.** The algorithm works for any network, but the complexity proof is only true for a ring (the proof for a general graph follows similar logic, but there are more cases and more events to consider).