6.5350 — Matrix Multiplication and Graph Algorithms

Class by Virginia Vassilevska Williams

Notes by Sanjana Das

Spring 2025

Lecture notes from the MIT class **6.5350** (Matrix Multiplication and Graph Algorithms), taught by Virginia Vassilevska Williams. All errors are my own.

Contents

§1 February 4, 2025

§1.1 Overview and logistics

The plan for this course is to introduce matrix multiplication as a computational problem. We'll work our way to designing MM algorithms. But in the first weeks, we'll see some equivalences and how to use MMs in graph algorithms contexts. And we'll also see how sometimes you can avoid it by allowing approximations.

We're going to be using Piazza; we'll probably use Gradescope for problem sets. The workload — we'll have 5 problem sets, roughly 2 weeks apart. Virginia planned each would be worth 5%, so together they'd be worth 25%.

The rest of your grade is on a project, towards the end. There's two types — learn a paper, or do your own research. For learn a paper, Virginia will publish a list of many papers related to MM or graph algorithms; you pick 1 or 2 to read, and read them and write a report. There's two, or really three, parts of the project. The first is a proposal — if you'll learn a paper you say which one you want and why. If you're doing research, you propose the problem you'll work on. If you'll do a resesarch project, it can be with a partner; learn a paper is individual. There will be more details as we go along. The major parts of the project are project presentations (in the last 2 weeks) and report (at the end, you write up all your findings — everything you learned from the paper, or all the approaches you took or everything you found in research). The proposal will be 5%; let's say the presentation and report are both 35%. If we need an extension, we should email.

§1.2 Matrix multiplication

Now we'll define matrix multiplication as a computational problem. (We'll abbreviate it as MM.)

Here you have some matrices A and B; let's say they're square matrices. Their entries are from some algebraic structure, typically fields ($\mathbb{R}$ or $\mathbb{C}$ or GF(2) or so on; sometimes we also talk about rings like $\mathbb{Z}$).

Problem 1.1 (Matrix multiplication)

- **Input:** $n \times n$ matrices A and B over some field K ($\mathbb{C}$, $\mathbb{R}$, GF(2), ...).
- **Output:** The matrix C where $C_{ij} = \sum_k A_{ik} \cdot B_{kj}$.

Question 1.2. How fast can we compute C?

When we talk about runtime, we need to specify how much operations cost. This depends on what field you're in. For $\mathbb{C}$ or $\mathbb{R}$, we'll assume the *real RAM model* of computation — where operations over $\mathbb{C}$ or $\mathbb{R}$ cost $O(1)$ time. This means addition and multiplication over the field cost constant time.

But if you're working over a field like $\mathbb{Q}$ (the rationals) or GF(w), we typically work over the *word-RAM*. This means constant-time operations are only over bounded-length numbers — if you look at a rational representation, operations are only constant over numbers with $O(\log n)$ bits. You have $O(\log n)$ bit integers; adding or multiplying such things takes constant time. (But because we can't represent complex or real numbers efficiently, we just assume operations with them take constant time.) SO here $O(\log n)$-bit operations are $O(1)$ time.

And then in order to figure out how long it takes to multiply matrices over your underlying field, we count how many operations ($\times$ and $+$) over that field we need in order to compute the product. In word RAM we also take the bit-lengths of the numbers into account — if they're too big then we'll have to pay a cost.

From now on, we're tpyically going to ignore this part about bit-length and talk about $\mathbb{C}$ or $\mathbb{R}$. But sometimes we'll come back to this, because with graph problems we'll sometimes have to worry about the sizes of numbers.

The question we really care about is determining something called ω.

> **Definition 1.3.** We use ω to denote the smallest real number such that $n \times n$ matrices can be multiplied using $O(n^{\omega+\varepsilon})$ time for every $\varepsilon > 0$.

The ε means we allow an overhead — ω doesn't have to be the *exact* exponent, as long as we can find an algorithm that goes as close to it as possible. (For example, you could have $n^{\omega}\mathsf{polylog}(n)$ or $n^{\omega}2^{\sqrt{\log n}}$ or so on. It's actually known that if ω is not 2, then it's a limit — if you can get an algorithm with $n^{\omega'}$, then you can get $n^{\omega''}$ where $\omega'' < \omega'$. This is kind of weird, but true — you can always make it better if $\omega \neq 2$.)

> **Fact 1.4 —** We have $\omega \geq 2$.

Why? You need to read all the numbers. But also, even if the matrices are sparse, you need to write the output. (Multiplying sparse matrices could still give you a dense matrix.) So in the worst case, you have to read the input and write the output, both of which can be pretty big.

For some reasons, lots of people think ω should be 2 — that there's an algorithm giving you almost linear algorithms for matrix multiplication. ('Linear' is because our input is $n \times n$.) One reason is we have such algorithms for related problems, like FFT. For polynomial multiplication you can get nearly linear algorithms using FFT, and this problem is similar.

Also, there's really no lower bounds — there's no better lower bounds than basically $2n^2$ or something, and even that was difficult to obtain.

That brings us to upper bounds. Up until 1969, people spent a lot of time trying to prove $\omega = 3$ — that the brute-force algorithm (there's n^2 choices of i and j, and n choices for k; you run over all of them and compute $\sum_k A_{ik}B_{kj}$) can't be beaten. This was proven for some limited types of algorithms, e.g., if you exclude subtraction and some other stuff.

But while trying to prove 3 is the best you can do, Strassen came up with an algorithm that achieved $\omega < \log_2 7 \approx 2.81$, which is notably smaller than 3.

Some years later, Pan generalized what Strassen was doing and proved $\omega < 2.79$. (Strassen's paper was called 'Gaussian elminiation is not optimal,' and this one was called 'Strassen's algorithm is not optimal.') There was lots of development bringing this down to a bit below 2.5; and then there was a breakthrough by Coppersmith and Winograd showing $\omega < 2.3754$. This was in 1986, and then nothing happened until 2010. In 2010, Stothers, a grad student in Edinburgh, wrote a thesis saying you can improve CW, but he didn't publish it because he went into finance; he showed $\omega < 2.374$. Then Virginia happened to have a very similar approach and happened to randomly read his thing; and her original approach got something worse than his, but after reading his she got 2.37288. Then Le Gall two years later got 2.37287. In 2021 Alman and Virginia got 2.37286.

Then in 2023, a completely different group — Duan, Wu, and Zhou — discovered you could do something different and got 2.37187. Then in 2024, Zhou was visiting, and Virginia, Xu, Xu, and Zhou improved this to 2.37156. The latest word is 2.37134 (Alman, Virginia, Xu, Xu, and Zhao).

As we can see, there's diminishing returns (we're chasing the fourth digit). This seems to be going to 2.37; there are various papers showing if you stick to these techniques, that's as far as you can get. (There's limitation papers showing if you use these approaches, you can't do much better.) So now we're searching for new approaches.

But this is not 2, so what gives? But anyways, that's why we're here... we're going to figure it out, right?

§1.3 Matrix multiplication vs. matrix inversion

For the rest of this lecture, Virginia will show us that matrix multiplication is equivalent to matrix *inversion*. This is kind of within the broad framework of showing matrix multiplication is one of the most central

problems in linear algebra — almost all linear algebraic operations you can think of can be solved using matrix multiplication, and many are actually equivalent to it. We'll exhibit such a reduction in both directions for matrix inversion; and at the end we'll talk a bit about Boolean MM, where we restrict the algebraic structure we work over.

Problem 1.5 (Matrix inversion)

- **Input:** an invertible $n \times n$ matrix A.
- **Output:** its inverse A^{-1}.

Virginia will show us two reductions. The first will be that if you can invert matrices in some running time $T(n)$, then you can also multiply matrices in roughly the same running time.

Claim 1.6 — If $n \times n$ matrix inversion is in $T(n)$ time, then $n \times n$ matrix multiplication is in $O(n^2 + T(3n))$ time.

(Technically the n^2 is not really needed because $T(n) \geq n^2$ — you have to read the input — but we'll put it here for convenience when doing the reduction.)

The second part only works over $\mathbb{R}$ or $\mathbb{C}$:

Claim 1.7 — If MM is in $T(n)$ time, then MI is in $O(T(n))$ time.

There's a slight caveat here about what the runtime function $T(n)$ is, but it's not too bad — you need $T(n) = n^2 \cdot f(n)$ where $f(n)$ is nondecreasing and at least a constant. (This basically says $T(n)$ is at least quadratic, which we need in order to read the input; and there's some nondecreasing function you tack on, which is basically without loss of generality since almost every runtime function we have is nondecreasing.)

So when we talk about $\mathbb{R}$ or $\mathbb{C}$, these two problems MM and MI are the same with respect to runtime. We'll prove Claim 1.6 first, because it's simpler; then we'll move to Claim 1.7, which has multiple steps.

§1.4 Inversion to multiplication

What we want to show is if there's an algorithm to invert matrices, we can use it to multiply matrices as well.

Here's what we do — say we have matrices A and B, and we want to multiply them to get AB, but we only know how to invert.

So we'll take A and B and create a matrix out of them, and then we'll invert that matrix, and we'll be able to read off AB from that matrix.

Here's our matrix — it'll be $3n \times 3n$ (where A and B are $n \times n$), and we'll split it into chunks of size n. We'll have

$$M = \begin{bmatrix} I & A & 0 \\ 0 & I & B \\ 0 & 0 & I \end{bmatrix}$$

where I is the $n \times n$ identity matrix.

Why is this matrix invertible? It's triangular with 1's on the diagonal, so its determinant is 1 (which is nonzero).

Claim 1.8 — We have

$$M^{-1} = \begin{bmatrix} I & -A & AB \\ 0 & I & -B \\ 0 & 0 & I \end{bmatrix}.$$

Proof. We can imagine multiplying out these matrices (using the blocks), and everything works. □

So this means we take this matrix M; to form it takes $O(n^2)$ time; then we invert it with my supposedly very fast inversion algorithm, which takes $T(3n)$ time. And after I invert it, I look at this top-right corner, and this gives us AB.

So that's the algorithm — once you can invert, you can also multiply.

Remark 1.9. For almost all natural functions, $T(3n) = O(T(n))$, so the 3 doesn't really matter.

§1.5 Multiplication to inversion

The second part is more technical. We'll take several steps.

We want to show if we can multiply matrices, we can also invert matrices in about the same time. So we're given some matrix A we want to invert, and we want to use a MM algorithm to slowly compute the inverse.

We're going to show that it suffices to be able to invert very special types of matrices — symmetric positive definite matrices. Then we'll show that if you can invert those, you can also invert arbitrary matrices in the same time. To get there, we'll define these special matrices and state a few properties, and then use them to get our algorithm for inversion.

§1.5.1 Symmetric positive definite matrices

Definition 1.10. A $n \times n$ matrix A is symmetric positive definite (SPD) if:

- It's symmetric, i.e., $A = A^\intercal$.
- It's positive definite, meaning that for every vector $x \neq 0$, we have $x^\intercal Ax > 0$.

The nice thing about these matrices is that:

Fact 1.11 — A SPD matrix A is always invertible.

One reason is that all its eigenvalues are positive. A simpler way to see this is that being invertible means for every nonzero vector x you have $Ax \neq 0$. So if it weren't invertible there would be a nonzero vector with $Ax = 0$, and then you'd have $x^\intercal Ax = 0$.

So these matrices are always invertible.

Fact 1.12 — For any invertible matrix A, $A^\intercal A$ is SPD.

So if I take any invertible matrix and multiply it by its transpose, I get a SPD matrix.

This is true because it's clearly symmetric —

$$(A^\intercal A)^\intercal = A^\intercal (A^\intercal)^\intercal = A^\intercal A.$$

And it's SPD because if I take any $x \neq 0$, I have

$$x^\intercal A^\intercal A x = (Ax)^\intercal (Ax) = \|Ax\|^2 > 0.$$

(It's not 0 because A is invertible, so $Ax \neq 0$.)

This is great. We'll see a few more properties in a bit. But this second property is already sufficient for us to show that if you can invert SPD matrices, you can also invert an arbitrary invertible matrix in the same time.

Claim 1.13 — Spupose $n \times n$ matrix multiplication is in time $T(n)$, and $n \times n$ SPD matrix inversion is also in $O(T(n))$ time. Then $n \times n$ matrix inversion (for arbitrary invertible matrices) can also be done in $O(T(n))$ time.

Proof. Here's the algorithm: Let's suppose we're given some $n \times n$ invertible matrix A (not necessarily SPD — it's an arbitrary invertible matrix). And now we want to invert it — we want A^{-1}.

What we'll do first is multiply it by its transpose — so we first compute $A^\intercal A$. This takes $T(n)$ time, because it's matrix multiplication.

Now, A is invertible, so $A^\intercal A$ is a SPD matrix; and we supposedly have an algorithm that can invert SPD matrices. So we compute the inverse

$$B = (A^\intercal A)^{-1}.$$

This is also $O(T(n))$ time, by assumption (because $A^\intercal A$ is SPD).

Now, what is B? We can also write

$$B = A^{-1}(A^\intercal)^{-1}.$$

So finally, how do I get A^{-1} out of this? We multiply it by $A^\intercal$ — so to finish, we compute

$$BA^\intercal = A^{-1}.$$

This is also matrix multiplication, so it takes $T(n)$ time.

So in three steps, we're done — we performed two matrix multiplications (one to convert to SPD, and one to get the output); and in the middle we did one inversion of a SPD matrix. And this gives us our reduction. □

Student Question. *Why did we need $T(n) = n^2 f(n)$ for nondecreasing f?*

Answer. This is technical; we'll need it in the second part, where we create the SPD inversion algorithm and analyze it.

§1.5.2 SPD matrix inversion

What remains to show is that if you can multiply matrices in $T(n)$ time, you can get SPD matrix inversion in that time as well.

So now we're going to use matrix multiplication to invert SPD matrices.

Claim 1.14 (Main) **—** If MM is in $T(n)$ time, then SPD matrix inversion is also in $O(T(n))$ time.

To prove this, we need some more properties of SPD matrices.

The next property is that if I take a SPD matrix and look at any *principle submatrix* — a square submatrix in the top-left corner — then that one is also SPD.

Fact 1.15 — Any principle submatrix of a SPD matrix A is also SPD.

The proof is that it's still symmetric, and you can pad x with 0's to get the same thing — if I want to show that $x^\intercal Mx > 0$ (where M is our submatrix), I just pad it with a bunch of 0's, and then I get the same output $x^\intercal Ax$; and because A itself is SPD, $x^\intercal Ax > 0$ (with the padded 0's); and because of that, $x^\intercal Mx > 0$. So this is basically by padding.

In particular, this means M is invertible; and if you had an algorithm to invert a SPD matrix, you could use it here on a smaller submatrix, which means you can recurse. (We'll see that in a second.)

The final thing is: Suppose A is a $n \times n$ SPD matrix and n is even. Then if you split A into chunks of size $(n/2) \times (n/2)$, as

$$A = \begin{bmatrix} M & B^\intercal \\ B & C \end{bmatrix},$$

we know that M is SPD. And we define this thing called the Schur complement.

Definition 1.16. The Schur complement is $S = C - BM^{-1}B^\intercal$.

Fact 1.17 — The Schur complement S is also SPD.

It's easy to see why S is symmetric; proving it's positive definite requries some thought, but it's not too bad (it's an exercise, which may or may not be on a pset).

Now we'll see an algorithm which uses recursion and matrix multiplication to invert any SPD matrix.

First, we'll assume A is $n \times n$ where n is a power of 2. If n is not a power of 2, then you can take n' to be the closest power of 2 igger than n (then $n' < 2n$, so this doesn't increase the matrix by too much), and you take

$$\begin{bmatrix} A & 0 \\ 0 & I \end{bmatrix}$$

where I has dimension $n' - n$. Then the inverse of thsi thing is a block matrix with A^{-1} and I. So if we can invert this, we can read off A^{-1} from the top-left corner. So because of this padding thing, we can always assume the input has dimension n which is a power of 2, without really increasing the problem size by more than a factor of 2.

So we'll assume A is a $n \times n$ SPD matrix, where n is a power of 2.

We'll call this algorithm $\mathsf{InvertSPD}(A)$.

The first thing we'll do is check if $n = 1$ — this means the matrix is just a scalar, so we just return $1/A$ (abusing notation a bit).

Otherwise, we can write A in blocks as

$$A = \begin{bmatrix} M & B^\intercal \\ B & C \end{bmatrix}$$

(as before). First we're going to invert M — so we compute $M^{-1} = \mathsf{InvertSPD}(M)$. This is recursion.

Now, we get to compute the Schur complement

$$S = C - BM^{-1}B^\intercal.$$

To compute this, we don't need to recurse at all — this is just matrix multiplication, which we assumed is in $T(n)$ time. So this part takes $O(T(n))$ time.

Now we claim that from S and M^{-1}, we can actually get A^{-1}. In order to do that, we're actually also going to invert S.

So we compute $S^{-1} = \mathsf{InvertSPD}(S)$, also using recursion.

Now we're going to output the inverse of the original matrix. And actually luckily for us, there's a formula for this!

Fact 1.18 — We have

$$A^{-1} = \begin{bmatrix} M^{-1} + M^{-1}B^\intercal S^{-1}BM^{-1} & -M^{-1}B^\intercal S^{-1} \\ -S^{-1}BM^{-1} & S^{-1} \end{bmatrix}.$$

So over the reals, as long as you have S^{-1} and M^{-1}, you can actually convert A^{-1} only by computing matrix products. (You can verify that this is the case.) Now we just multiply these things and add them up, and we get our inverse of A.

So the point is that we can compute the above thing with a constant number of matrix multiplications and additions; this is basically $O(T(n))$.

If you believe this, then this recursive algorithm is correct, and we just have to analyze its runtime. Do we believe it? You have to verify the formula, but it's not too bad. For example, if we take the top row and the left column, we get

$$I + BS^{-1}B^\intercal M^{-1} - B^\intercal S^{-1}BM^{-1},$$

and it works out.

So now let's analyze runtime — let $T_{\mathsf{inv}}(n)$ be the runtime of this algorithm on a $n \times n$ matrix. We're going to write a recurrence in terms of $T_{\mathsf{inv}}(n/2)$ and matrix multiplication.

We claim

$$T_{\mathsf{inv}}(n) \le 2T_{\mathsf{inv}}(n/2) + O(T(n)).$$

Why? We do two recursive calls on matrices of size $n/2$, and then a constant number of matrix multiplications taking $O(T(n))$ time. (The two recursive calls are one to invert M and one to invert S; and the rest — forming S is $O(T(n))$, and we do a constant number of MMs afterwards.)

So now we have this recurrence, and we can write it out as

$$T_{\mathsf{inv}}(n) \lesssim \sum_i 2^i T(n/2^i)$$

(on the ith step we have 2^i calls, where the matrix dimension has dropped by a factor of 2^i). We're going to ignore the O for convenience; then using $T(n) = n^2 f(n)$, we get

$$n^2 \sum \frac{2^i f(n/2^i)}{(2^i)^2}$$

(where $T(n/2^i) = (n/2^i)^2 f(n/2^i)$). And f is nondecreasing, so $f(n/2^i) \le f(n)$. And with the powers of 2, we have $\sum_i 2^i/4^i = \sum 1/2^i$, which is a constant. So we get

$$T_{\mathsf{inv}}(n) \lesssim n^2 \cdot \sum \frac{1}{2^i} \cdot f(n) = O(n^2 f(n)),$$

which is exactly what $T(n)$ is.

Remark 1.19. We did this $T(n) = n^2 f(n)$ assumption so that within constants it's actually the same; if it doesn't have this form then you might get some log's.

Here we used SPD matrices, which make sense over the reals; you can extend it to complex numbers by looking at Hermitian matrices. For finite fields, this is not the same; Virginia isn't sure whether it's known that the two are equivalent.

You can prove such things for other linear algebraic operations; we'll give you something on the pset that you can play around with.

But now in the remaining 20 minutes, we'll talk about *Boolean* matrix multiplication.

> **Student Question.** *Where did we use the SPD property?*
>
> **Answer.** We needed M and S to be invertible; that's where we use it.

§1.6 Boolean matrix multiplication

§1.6.1 Motivation

These algorithms Virginia mentioned for matrix multiplication have nice exponents, but they're somewhat impractical — when we develop them we use very heavy machinery that only works for very, very large matrices (because we take things in the limit). So almost none of these algorithms are used in practice, except Strassen's. There are ways to mitigate this issue, but there's extra overhead.

Because of this, people really want practical 'combinatorial' algorithms. This isn't well-defined, but there's some vague notion of what practical means.

Vaguely, you want to take your input and somehow preprocess it so tha tsomehow multiplication becomes faster using a bunch of lookup tables or something about a decomposition of your input. And typically people don't want you to use subtraction, which is very crucial to the earlier algorithms — in these algorithms you take linear combinations and cleverly recombine them. But that causes lots of impracticalities, and people don't want that in practice. Hopefully you should just use the numbers from your input without changing them too much — you keep them the same but somehow decompose them into pieces that can be multiplied fast. That's roughly the idea of combinatorial algorithms; but it's not well-defined.

We're going to see one combinatorial algorithm for multiplying Boolean matrices; it also works for integer matrices, as long as the integers are 0's and 1's.

§1.6.2 The problem

Problem 1.20 (BMM)

- **Input:** matrices A and B with entries in $\{0, 1\}$.
- **Output:** $C_{ij} = \bigvee_k (A_{ik} \wedge B_{kj})$.

BMM looks much simpler than MM in general, and it can be solved by embedding into the integers. But it potentially might have much faster algorithms, and lots of work has been done in trying to get them by coming up with alternative 'combinatorial' algorithms.

BMM turns out to be sufficient for almost all the applications within graph algorithms — those don't generally care about the *number* of witnesses (the number of k with $A_{ik}B_{kj} \neq 0$), just whether there *is* some k. So that's why we care about this problem. And it turns out there's much easier ways to solve it in some cases.

§1.6.3 History

There's something you can do called the *our Russians algorithm*. The people who came up with this algorithm were supposedly not all Russians, but they were from the Soviet Union at the time – Arlazarov,Dinic, Kronrod, and Faradzev.

For $n \times n$ matrices, the runtime of this algorithm is $O(n^3/(\log n)^2)$. (The square depends on the model of computation; this is over word RAM.) This also works for integer matrix multiplication as long as A and B have only 0's and 1's.

Virginia will give us that algorithm; but the best-known BMM algorithm not using the MM technology from earlier is actually faster. It's from this year (2024), and it's by Abboud, Fischer, Kelley, Lovett, Meka; it achieves $O(n^3/\exp((\log n)^{1/7}))$. This $\exp((\log n)^{1/7})$ is better than any polylogarithm, so it's much better than $(\log n)^2$. This only works for BMM (not integer matrix multiplication). And it is not known whether you can get $n^{2.999}$ without using the techniques of MM from before (staying within the Boolean setting). This is a major open problem. There's some sort of fake conjecture:

> **Conjecture 1.21** (BMM conjecture) — No $O(n^{3-\varepsilon})$ time combinatorial BMM algorithm exists.

The reason this is a 'fake' conjecture is that we don't know what 'combinatorial' means, so it's not well-defined.

§1.7 The four Russians algorithm

We'll quickly outline the four Russians algorithm.

§1.7.1 Lookup tables

We'll make an assumption: that if you have a lookup table of $\mathsf{poly}(n)$ size indexed by $O(\log n)$-bit integers, then I can look up an entry in the table in constant time. This will be part of our model — that if T is a $\mathsf{poly}(n)$-sized table indexed by $O(\log n)$-bit integers, then we can look up $T(k)$ for any key k in $O(1)$ time. (In the word RAM model you can basically do this.)

What does this assumption let us do? For example, I can take all possible roughly $\log n$ bit strings, and precompute their componentwise OR or their inner products, and store them in a table. And now for any tiny $\log n$-bit strings, I can look up their inner product or their componentwise OR in constant time.

For example, if we have some small $\varepsilon > 0$, I can precompute $x \vee y$ (this $\vee$ is componentwise OR, and for the Boolean case; you could also do $x + y$ over integers if you want) for all $x, y \in \{0,1\}^{\varepsilon \log n}$ and store it in T_{OR}. Now if I want $x \vee y$, I can just look it up as $T_{\mathsf{OR}}(x, y)$ in constant time. I can do the same thing for inner product — we store $x \cdot y$ in $T_{\mathsf{IP}}(x, y)$.

How much time does the precomputation take? There are $2^{\varepsilon \log n} = n^{2\varepsilon}$ pairs (x, y), and then there's $\log n$ to compute them. So it's cheap — and I precompute these and store them, and then it's very cheap to look them up.

§1.7.2 Preprocessing

Second, let's say I'm given a Boolean matrix. I'm going to preprocess it so that multiplying it by *column* vectors will be very cheap.

So let's say A is a $n \times n$ Boolean matrix. We're going to preprocess it so that Ax (for Boolean vectors x) is cheap.

The way we do this is we take A and split it into tiny blocks, each of size $\varepsilon \log n$, which we call $A_{a,b}$ (this is the (a,b)th block). This is an $\varepsilon \log n \times \varepsilon \log n$ matrix, and there are $(n/\varepsilon \log n)^2$ choices for a and b. I'm going to take any such matrix, and I'm going to precompute its Boolean product with all vectors of length $\varepsilon \log n$, and store that in a lookup table. So for very a and b, and every $y \in \{0,1\}^{\varepsilon \log n}$, I precompute $A_{ab} \cdot y$ and store it in a table as $T_{ab}(y)$. Now the product of A_{ab} with any small vector can be looked up in constant time.

The runtime for this precomputation is very cheap — the number of a and b is $(n^2/\varepsilon \log n)^2$, and multiplying a $(\varepsilon \log n) \times (\varepsilon \log n)$ matrix by a small vector is $(\varepsilon \log n)^2$. And you also multiply by the number of these y's; so you get

$$2^{\varepsilon \log n} \cdot \frac{n^2}{(\varepsilon \log n)^2} \cdot (\varepsilon \log n)^2 = n^{2+\varepsilon}.$$

So with this, I can preprocess to look up the Boolean product of any such block A with any Boolean vector of length $\varepsilon \log n$.

§1.7.3 The conclusion

Now given this, we can multiply A by *any* given vector quickly, as follows. You have your matrix A (which is $n \times n$), and you're given some column vector x. And you have A chunked into tiny pieces; and you correspondingly chunk x into the same $\varepsilon \log n$-sized pieces (so we have chunks A_{ab} and correspondingly x_b). Now we can multiply Ax as follows — let's call the output v. We start by setting v to the all-0's. Then for all a and b, we compute

$$v_a \coloneqq v'_a \vee (A_{ab} \cdot x_b)$$

(where v'_a is the original thing). And this $A_{ab}x_b$ can be looked up in constant time in $T_{ab}(x_b)$, which we precomputed. And this OR can also be looked up in $O(1)$ time in T_{OR}. So all the operations inside this loop are constant time. This means the runtime is just going over all a's and b's, so the runtime is just the number of a's and b's, which is $n^2/(\log n)^2$.

Now to multiply AB, we just multiply A by all the column vectors of B; the total runtime is going to be this times the number of column vectors (which is n) plus the preprocessing time.

(The preprocessing time is superquadratic, but it only happens once, so if you're multiplying by many column vectors then it's negligible.)

So this is the four Russians algorithm. When it was originally published, the $\log^2$ wasn't there — they had published it with just a log. The reason is they were trying to avoid this lookup table business — trying to not assume you can look things up in constant time. Also, for preprocessing, you don't need Booleans — you just need the vectors to be over 0 and 1, so that you have a constant number of choices for each entry (and then you can do the same thing).

§2 February 6, 2025

We'll have video recordings; they start at 11:03, but class doesn't start until 11:05. Yael and Virginia will finalize the pset and release it tonight, and then we have about two weeks to do it. All the lecture notes will be on Piazza; last lecture's notes are already there. If you see any typos, please let Virginia know.

Today, we're going to talk about an equivalence between transitive closure and Boolean matrix multiplication, which basically means they're the same problem — if you happen to find a very fast algorithm for BMM (even if typical matrix multiplication doesn't have very fast algorithms, maybe BMM does), you'll be able to solve transitive closure as well.

Then we'll talk about all-pairs shortest paths (APSP). It's typically taught in undergraduate algorithms (e.g., 6.1220), so we may have seen it before. But today we'll discuss undirected unweighted graphs in

particular. We'll show that there the problem can be solved using MM; on the pset we'll actually show it's equivalent to BMM, so it's the same as transitive closure.

§2.1 Transitive closure

First, what's transitive closure (TC)?

Problem 2.1 (Transitive closure)

- **Input:** A directed graph G (with n nodes).
- **Output:** An $n \times n$ matrix T such that

$$T_{uv} = \begin{cases} 1 & \text{if } u \rightsquigarrow v \\ 0 & \text{otherwise.} \end{cases}$$

This notation means there's a path from u to v; so we want to output all pairs' reachability (which vertices can reach others). Today we'll show that if TC has a $T(n)$ time algorithm, then BMM does too (with roughly the same runtime); and if BMM has an algorithm, then so does TC.

Like last lecture, we'll assume our runtime functions $T(n)$ are 'nice,' in the following sense:

- $T(O(n)) = O(T(n))$. (For example, all polynomials have this property.)
- $T(n/2) \leq T(n)/4$. (This basically means T is at least quadratic.)

§2.1.1 Transitive closure to BMM

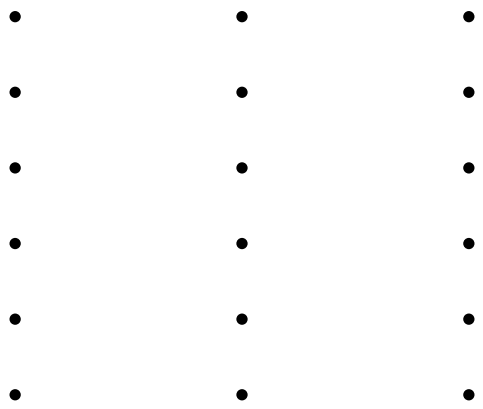
Let's start with the simpler claim.

Claim 2.2 — If transitive closure on n-node graphs is in $T(n)$ time, then Booleean matrix multiplication on $n \times n$ matrices is in $O(T(n))$ time.

So we assume we have a very fast algorithm for solving TC on a directed graph. Now we're given two $n \times n$ matrices, and we want to compute their Boolean product — we're given $\{0,1\}$-matrices A and B, and we want to output C where

$$C_{ij} = \bigwedge_k (A_{ik} \wedge B_{jk}).$$

Proof. There's a standard encoding of matrix products as a graph product: if we have Boolean matrices A and B, you create a graph as follows. You start with putting n vertices on the left, n in the middle, and n on the right. The vertices on the left are the rows of A (there's a vertex for every row in A). The vertices on the right are the columns of B. The vertices in the middle are the columns of A and rows of B together (because in our matrix product, the two k's are the same).



Then you encode A between the left and middle part, and B between the middle and right. How to encode? For every row i and column k of A, we look at i on the left and k in the middle. If $A_{ik} = 1$, then we put a directed edge $i \to k$; otherwise (if it's 0) we don't. Then you do the same for B — we look at k in the middle and j on the right, and we put an edge $k \to j$ if and only if $B_{kj} = 1$.

Now $i \in L$ can reach $j \in R$ only if there is some midpoint k such that $A_{ik} = 1$ and $B_{kj} = 1$.

So we've created a graph G that has $3n$ nodes, and in this graph, if we can compute the transitive closure, then we can check for every $i \in L$ and $j \in R$ whether there is a path or not; so we can compute this output in $O(n^2)$ additional time. Creating this graph is $O(n^2)$ time, and reading off the output is also $O(n^2)$; and then there's this $T(3n)$ time to compute the transitive closure (because $3n$ is the number of vertices). We said that T is nice, so this is $O(T(n))$.

So our total runtime is $O(T(n)+n^2)$. And the assumption $T(n/2) \leq T(n)/4$ means $T(n)$ is at least quadratic, so the n^2 gets absorbed. (Anyways, you need $T(n) = \Omega(n^2)$ for transitive closure because you need to *read* the graph.) □

§2.1.2 A warmup

Now we're going to try to do the opposite.

> **Claim 2.3 —** If BMM is in $T(n)$ time, then TC is in $O(T(n))$ time.

Before we do this, as a warmup, we'll see a simple way to get $T(n) \log n$ time.

We've got an algorithm that can multiply Boolean matrices, and we want to compute the transitive closure of our directed graph G. So first we compute the adjacency matrix A, where

$$A_{ij} = \begin{cases} 1 & (i,j) \in E \\ 0 & \text{otherwise.} \end{cases}$$

Now we look at $(A \vee I)$ (which is basically A, except that we also put 1's on the diagonal). Then

$$(A \vee I)_{ij} = \begin{cases} 1 & \text{if } d(i,j) \leq 1 \\ 0 & \text{otherwise} \end{cases}$$

(just by definition). Now imagine we take this thing to the ℓth power; then we claim

$$(A \vee I)^{\ell}_{ij} = \begin{cases} 1 & d(i,j) \leq \ell \\ 0 & \text{otherwise.} \end{cases}$$

(You can show this through induction — we already have the base case. When you multiply twice, you're asking, does there exist k such that $d(i,k) \le 1$ and $d(k,j) \le 1$? And then we actually get that whenever there's a path of length at most 2, you'll always find such a k, and vice versa; so the second power just records, is the distance at most 2? And you can show the general case by induction. It's an exercise, but it should be somewhat clear why it's the case.)

Claim 2.4 — We have $T(G)_{ij} = (A \vee I)^{n-1}$.

That's because $n-1$ is the maximum possible length of a shortest path — without loss of generality, the shortest path from i to j is simple (if you have a path that has cycles, you can kill them off and get a shorter path). This means the shortest paths are of length at most $n-1$; so the distance between any two vertices is either infinite or at most $n-1$.

So once you compute the $(n-1)$st power, for every i and j you know whether there's a path or not (you don't have to go to ∞).

And not only this, but this Boolean product of matrices is also associative. So we don't have to multiply one by one; we can do successive squaring. This means to get to the $(n-1)$st power, we can just do $\log n$ matrix multiplications instead of $n-1$. (This is not true for *all* matrix products — later on in the class we'll see examples where you can't — but here you can.) So you do successive squaring — you compute

$$(A \vee I)^{2^\ell} = (A \vee I)^{2^{\ell-1}} \cdot (A \vee I)^{2^{\ell-1}},$$

and you keep doing this until $2^\ell \ge n-1$, and then you're done. This is $O(\log n)$ iterations, each costing you $T(n)$; so you get $O(T(n) \log n)$.

§2.1.3 Strongly connected components

This is just a warmup; but we claimed that you actually don't need the log. To do that, we'll actually look at the graph and the fact that it is a graph, and we'll do something with it.

Our input is a directed graph. We'll consider its *strongly connected components* — this is a maximal set where any two vertices in it can both reach each other. So a SCC is a subgraph that's maximal (i.e., you can't add any more vertices) such that inside it, for every pair of vertices, there's a path in both directions.

Definition 2.5. A strongly connected component of G is an inclusion-maximal subgraph C such that for all $u, v \in C$, we have $u \rightsquigarrow v$ and $v \rightsquigarrow u$.

Example 2.6

If we have a graph $1 \to 2 \to 3 \to 4 \to 1$, $1 \to 3$, $2 \to 5$, $4 \to 5$, $5 \to 6$, $4 \to 6$, then the SCCs are $C_1 1234$, $C_2 = 5$, and $C_3 = 6$.

A second thing is, it's well-known that you can compute all the SCCs of a graph in *linear* time (in the number of edges and vertices of the graph).

Fact 2.7 — There's an $O(|E| + |V|) \le O(n^2)$ time algorithm to compute all the SCC's.

(Tarjan was involved; it uses DFS.)

Next, once you have the SCCs, you can squeeze each one of them into a single vertex. Let's say I collapse every single one of them into a single vertex; in this example, what will happen is I get C_1, C_2, and C_3. And what I do is if there was some edge from C_1 to C_2, I put it in here. (If there are two edges, I just keep one

of them; I remove all duplicates.) So you get a graph whose vertices are SCCs, and you have edges between them.

What's special about this graph is it's *acyclic* — it's a *directed acyclic graph*.

> **Definition 2.8.** The SCC-DAG is the graph whose vertices are SCCs, and with edges betwen two SCCs if the original graph had an edge between them. (In other words, you collapse SCCs into singletons.)

(It's a DAG because if there were a cycle, then we could have made a bigger strongly connected component.)

The next thing is, if I look at a strongly connected component, and the part of the transitive closure matrix on that connected component, what is it? It's all 1's — every pair of vertices is reachable in both directions. So if we have a SCC, we know that part of the transitive closure is all 1's. For example, here we have a 4×4 block of 1's corresponding to 1234. So the vertices inside a SCC are basically the same on the inside. And how they relate in terms of reachability to other vertices in the graph is also the same — if 1 can reach 5, so can everything else in its component.

So we might as well solve transitive closure in the SCC-DAG. The moment we get that, we'll have some matrix; it'll be a 3×3 matrix. But we know everything in C_1 is the same, so then we can uncollapse that matrix to get the original. For example, here our transitive closure matrix (indexed by C_1, C_2, C_3) is

$$\begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix}.$$

Then to get to the transitive closure of the full graph, all we do is uncollapse C_1 by making it into a 4×4 thing, replacing every 1 in its row and column by a matrix of 1's (and the 0's by matrices of 0's).

So all we have to do is compute the SCCs, make this SCC-DAG, solve TC on a directed acyclic graph, get the matrix out of it, and then uncollapse — get from the small matrix from the SCC-DAG and make it into a bigger matrix, where every 1 for a component gets blown up into its all-1's matrix.

So to recap:

> **Conclusion 2.9.** If you can compute TC on a n-node DAG in $T(n)$ time, then computing TC on an n-node *arbitrary* directed graph is also $O(n^2 + T(n))$ time.

(The n^2 will be subsumed in the $T(n)$, since $T(n) = \Omega(n^2)$; but it's to compute the SCCs and extract the final transitive closure matrix in the end.)

§2.1.4 Transitive closure on DAGs

The next thing we'll do is try to actually get an algorithm that will compute transitive closure on a n-node DAG, given that we can multiply Boolean matrices in $T(n)$ time.

> **Claim 2.10 —** If BMM is in $T(n)$ time, then TC on a DAG is in $O(T(n))$ time.

To do this, we need to remember some nice properties of DAGs. Do you remember any nice properties that one could possibly exploit?

If you have a DAG, you can *topologically sort* it:

> **Definition 2.11.** A topological order of a DAG is a sorted order of the vertices $v_1, v_2, \ldots, v_n$ such that every edge goes from left to right (i.e., if $(v_i, v_j) \in E$, then $i < j$).

So the topological order tells you there are no edges in the opposite direction. But it doesn't tell you whether v_2 can reach v_{100}; so this doesn't actually give you reachability.

But it does give us a very nice sorted order of the vertices. In particular, suppose I have my DAG and I compute a topological order

Fact 2.12 — A topological order on a DAG can be computed in $O(V+E) \leq O(n^2)$ time.

And now let's look at the adjacency matrix, with the vertices sorted in that order (so we have $v_1, v_2, \ldots, v_n$ in the rows and columns). Then this matrix is *upper triangular* — below the diagonal, it's all-0's (because you cannot have an edge from a vertex later in the order to a vertex earlier). So the adjacency matrix is actually upper triangular.

And this is also true of the transitive closure matrix — it's also upper triangular. (It's also true of A^ℓ for any ℓ.)

This will give us some ideas for hwo to continue.

Let's look at this matrix A again. Again, we're going to assume that n is a power of 2 (if it's not, you can always make it a power of 2). So we have our matrix A, and we'll split it into pieces of size $n/2$. We know the part on the bottom-left is 0. The parts on the top-left and bottom-right are also upper triangular; so we get

$$A = \begin{bmatrix} B & Q \\ 0 & C \end{bmatrix},$$

where B and C are upper triangular. We'll also call the vertices corresponding to these $n/2$ things as L and R; so in terms of the topological order, the first $n/2$ vertices are L and the last $n/2$ are R.

In here, B is the adjacency matrix of the graph restricted to L; and C is the adjacency matrix of the graph restricted to R. And Q just tells me the edges going between L and R (across the boundary).

Now, I want to express the transitive closure of A in terms of the transitive closures of B and C and Q. For shorthand, let's say for a matrix S, we use S^* to denote its transitive closure.

So now we have

$$A^* = \begin{bmatrix} B^* & B^*QC^* \\ 0 & C^* \end{bmatrix}$$

Of course, A^* is upper triangular, so the bottom-left is 0. On the top-left, we have B^* — the piece of the graph inside L, you can never go into R and come back, so the transitive closure of L is actually just the transitive closure of its adjacency matrix. Similarly, if we're in R, we can't go out of R; so the transitive closure of R doesn't depend on anything in L. And for the top-right, any path from L to R goes somewhere within L, then has to exit L through an edge in Q, and then it stays in R and can't go back. (Here B^*QC^* is the Boolean matrix product.)

(To restate what happens with B^*QC^*, we're in L and then there's R, and there's a bunch of edges between them described by Q. Say I have $i \in L$ and $j \in R$; what are the paths between i and j? All the paths are going to involve going inside L somewhere to some vertex k, then jumping to R through one of the edges in Q to some other vertex r, and then taking a apth from r to j. The paths on the left and right are described by B^* and C^*, and the single edge is described by Q.)

Given this, we get an algorithm to compute the transitive closure of a DAG: let's say n is a power of 2, and we want to compute $\mathsf{TC}(A)$. There's some base case, where $n = 1$; then we just return 1 (any vertex can reach itself). Otherwise, you split into halves; you compute $B^* = \mathsf{TC}(A_{LL})$ and $C^* = \mathsf{TC}(A_{RR})$, and you compute the Boolean matrix product B^*QC^*. And then you return the above block matrix.

The runtime recurrence for this is

$$t(n) = 2t(n/2) + O(T(n))$$

(for the two transitive closure calls and the BMM step, which involves two Boolean matrix multiplications). And we saw that if $T(n)$ has the nice property $T(n/2) \leq T(n)/4$, then this solves to $t(n) \leq O(T(n))$ — so we don't get any logs. (This is the same exact recurrence from last class.)

So we've killed the log; it's gone.

§2.2 All-pairs shortest paths

Now we're going to move on to shortest paths, which is potentially a harder problem. But it turns out for undirected graphs, it's actually equivalent.

Problem 2.13 (APSP)

- **Input:** a graph G (which may be directed or undirected, and weighted or unweighted).
- **Output:** the $n \times n$ matrix d, where $d(u, v)$ is the shortest paths distance from u to v.

You can have weights on vertices or edges, and the weights can be positive or negative, as long as there are no negative cycles (if you had negative cycles, then the answer would be $-\infty$).

So we're not computing the paths, just the distances; we'll later discuss how to compute the paths.

What's known?

The runtime really depends on whether you're directed or undirected, and weighted or unweighted. We're going to ignore the dependence on the number of edges for now (we'll discuss that later), and just talk in terms of n.

For *weighted* graphs, the best known algorithm is $n^3/\exp(\sqrt{\log n})$, by Ryan Williams from 2014. Here it doesn't matter whether the graph is directed or not (it's equivalent for weighted graphs).

Student Question. *What does this algorithm look like?*

Answer. This algorithm is very complicated. Before this there were algorithms that shaved polylogs; this one used the polynomial method and techniques from circuit lower bounds, where you express circuits in an algebraic way and then use rectangular MM to solve these problems and shave a bit more. But still this is not $n^{2.9}$, and there's some hypothesis that you can't get that.

For unweighted graphs, we have two different runtimes depending on whether the graph is directed or not. For *undirected* graphs, you can get $O(n^\omega \log n)$. For the rest of the class, we won't care about the log factors, so we'll just write this as $\widetilde{O}(n^\omega)$. And for *directed*, the best-known runtime is $O(n^{2.53})$. The undirected case is due to Seidel, and the directed case is due to Zwick.

This ω is the exponent of *Boolean* matrix multiplication; the 2.53 is an exponent which you get by looking at rectangular matrix multiplications and doing some tricks. If $n \times n$ matrix multiplication actually has a fast algorithm — if $\omega = 2$ — then Zwick's algorithm is actually just $O(n^{2.5})$. There's actually reasons to believe that you can't improve over 2.5 for directed graphs — there are papers showing if you can, you get some very interesting consequences. So there's actually a conditional separation between directed and undirected graphs (for the unweighted case).

In the next 30 minutes, we're going to see Seidel's algorithm; but instead of using BMM, we'll use *integer* matrix multiplication. On the pset we can do the Boolean case.

§2.3 APSP in unweighted undirected graphs

Here you have a graph $G = (V, E)$ with n nodes that's given to you. And we want to compute distances.

So we're going to take the adjacency matrix A, just like before. And let's look at the matrix

$$A' = A^2 \vee A$$

(where A^2 denotes the Boolean product). This A' describes a new graph — it's the adjacency matrix of some graph which we'll call G^2. Let's look at what this graph looks like. Say we start with some undirected graph with edges 12, 23, 34, 41, 35. Now, the OR of A means the edges of G^2 contain the edges of G; so we draw those in. But there's also A^2. This means you get an edge for every length-2 path in the original — for every length-2 path, there's an edge betwen its endpoints. So we get the edges 13, 24, 25, 45.

So G^2 contains the original G, plus edges representing paths of length 2.

Next, we write $d_{G^2}(u, v)$ for the distance from u to v in G^2.

Claim 2.14 — We have $d_{G^2}(u, v) = \lceil d(u, v)/2 \rceil$.

Proof. Let's say I take my shortest path in G from u to v, and its length is $d(u, v)$. Then for every two vertices, there's a yellow edge in G; so I get to skip them. And if it's odd, the last edge is left over. So this shows $d_{G^2}(u, v) \leq \lceil d(u, v)/2 \rceil$. But also, it's *at least* $\lceil d(u, v)/2 \rceil$. Why? Let's look at the path in G^2. Every single edge in this path could at most represent a 2-path in the original graph; so because of this, $2d_{G_2}(u, v) \geq d(u, v)$. This means $d_{G_2}(u, v) \geq d(u, v)/2$; and it's an integer, so we can put a ceiling. □

§2.3.1 An 'algorithm'

So we have this very simple thing. Now we're going to give an algorithm with a caveat, that will be able to extract $d(u, v)$ given $d_{G^2}(u, v)$. It'll be recursive, but it'll have this caveat that we need to be able to tell whether $d(u, v)$ is odd or even; and then we'll try to figure out how to do that.

(So far, everything we've said would work just as well in directed graphs; we haven't done anything undirected yet.)

So here's an APSP 'algorithm' — suppose we have our adjacency matrix A. There's some base case, which we'll discuss in a little bit. Otherwise, we first compute $A' = A^2 \vee A$ and $d_{G^2} = \mathsf{APSP'Alg'}(A')$. Now for all $u, v \in V$, if $d(u, v)$ is even, then $d(u, v) = 2d_{G^2}(u, v)$; else $d(u, v) = 2d_{G^2}(u, v) - 1$. And then we return d.

There's a caveat here — we assume G is connected. If G is not connected, you first compute the connected components and aply this 'algorithm' on each connected component separately.

The first thign is, suppose we can do this parity check — I somehow know whether the distance is even or odd. When can I terminate this algorithm? Unlike before — in transitive closure, n was decreasing. Here n is not decreasing — the vertices of G^2 are the same. But something else is decreasing. The largest distance roughly halves, because all the distances get divided by 2. So in particular, the diameter roughly halves in each step. So even though n doesn't decrease, the distances do. And in $O(\log n)$ iterations, the diameter of your graph is going to be 1.

Now, how can I tell when the diameter has become 1? Everything has been connected to everything, so it's all 1's (except on the diagonal). So that defines our base case — if $A = J - I$, then we return A (because the distance is 1 if there's an edge and 0 if you're the same vertex).

(The undirected part is not super important yet, but it's going to be super important for the parity check.)

This isn't really an algorithm, because we don't know how to check whether distances are even or odd. But if we *did* know how to check, then this would run in the time it takes to multiply matrices, times $\log n$. This is just BMM — we don't need integer multiplication yet. But when we implement the parity check in this lecture, we will use integer MM.

So if the parity test can be done in $\widetilde{O}(n^\omega)$ time, then APSP is in $\widetilde{O}(n^\omega)$ by this algorithm (replacing the parity test in our description with an actual algorithm for it).

§2.3.2 The parity test

What's going to save us here is that we actually have access to d_{G^2}. Given the distances in G^2, we'll be able to figure out whether distances in G are even or odd. (If we didn't have this information, it's not obvious what to do; but given this, we will be able to do it.)

So we're given d_{G^2}; and we want to know, for all u and v, if $d(u,v)$ is even or odd — this is our problem now.

We're going to set this up a little bit — consider some shortest path in G. So we have a node i and a node j, and this si a shortest path in G. And consider any neighbor k of j. What do we know about $d(i,k)$ vs. $d(i,j)$? For the first thing, we know

$$d(i,k) \leq d(i,j) + 1.$$

What else do we know? We know

$$d(i,k) \geq d(i,j) - 1.$$

Here we're using the fact that the graph is undirected — so the triangle inequality holds in both directions, because $d(i,j)$ is the same as $d(j,i)$. However, this gives us something very interesting:

Claim 2.15 — If $d(i,j) \equiv d(i,k) \pmod 2$, then $d(i,j) = d(i,k)$.

So because they're so close, if they're both even or both odd then they're actually the same.

Next, suppose $d(i,j)$ is even and $d(i,k)$ is odd, so they have different parity. Then we want to say something about them in terms of d_{G^2}.

First, we have

$$d_{G^2}(i,j) = \frac{d(i,j)}{2}$$

(there's a ceiling, but $d(i,j)$ is even, so this is actually exact). Now, $d(i,k)$ is odd, and $d(i,j) \leq d(i,k) + 1$; so we get

$$d_{G^2}(i,j) = \frac{d(i,j)}{2} \leq \frac{d(i,k)+1}{2} = \left\lceil \frac{d(i,k)}{2} \right\rceil = d_{G^2}(i,k).$$

And in the other direction, if $d(i,j)$ is odd and $d(i,k)$ is even, symmetrically we get $d_{G^2}(i,j) \geq d_{G^2}(i,k)$.

And finally, there's one more thing. In this last case where $d(i,j)$ is odd and $d(i,k)$ is even, we'll look at i and j, and we'll look at the node k' which is right before j on the shortest path. And let's say $d(i,j)$ is odd. Then $d(i,k')$ is even, because $d(i,k') = d(i,j) - 1$ (it's right before j on the shortest path).

So then we look at $d_{G^2}(i,j)$. We have

$$d_{G^2}(i,j) = \frac{d(i,j)+1}{2} = \frac{d(i,k')+2}{2} > \frac{d(i,k')}{2} = d_{G^2}(i,k').$$

Where are we going with this? We're going to take all these $d(i,k)$'s and sum them all up. So we fix i and j, and consider

$$\sum_{k \in N(j)} d_{G^2}(i,k)$$

(where $N(j)$ denotes the neighbors of j). We know that when $d(i,j)$ and $d(i,k)$ are the same parity, their distances are exactly equal; so their distances in G^2 are also equal. If they're different parities, then you have different inequalities here. Whenever $d(i,j)$ is even and $d(i,k)$ is odd, we get the inequality

$$d_{G^2}(i,j) \leq d_{G^2}(i,k).$$

And in the opposite direction, you have the opposite inequality. And in the opposite direction, we *also* saw that there was some k' where the inequality was strict! So we have

$$\sum_{k\in N(j)} d_{G^2}(i,k) \begin{cases} < \deg(j)\cdot d_{G^2}(i,j) & \text{if } d(i,j) \text{ odd} \\ \geq \deg(j)\cdot d_{G^2}(i,j) & \text{if } d(i,j) \text{ is even.} \end{cases}$$

(If $d(i,j)$ is odd, then we get $d_{G^2}(i,j) \geq d_{G^2}(i,k)$ for all k, with a strict inequality for at least one k'; and otherwise we get weak inequalities in all the other directions.)

So now we have a parity test — if we can compare this sum with $\deg(j)d_{G^2}(i,j)$, then we can tell whether $d(i,j)$ is even or not.

So now the question is, for every i and j, how do we compute this quantity $\sum_{k\in N(j)} d_{G^2}(i,k)$ (knowing what d_{G^2} is)?

We can multiply d_{G^2} by the adjacency matrix!

So this is Seidel's algrithm to implement the parity test (for d): we multiply $d_{G^2}\cdot A$ (using *integer* matrix multiplication), because

$$(d_{G^2}A)_{ij} = \sum_k d_{G^2}(i,j)A(k,j) = \sum_{k\in N(j)} d_{G^2}(i,k)$$

(since $A(k,j)$ is 1 only when kj is an edge). And therefore, using integer MM given d_{G^2}, we can compute the left-hand side. And then for every i and j, we know $\deg g$, so we can compute $\deg(j)d_{G^2}(i,j)$, and the comparison is very simple.

So in our original 'algorithm,' after we got d_{G^2}, we compute $d_{G^2}\cdot A$, and then we implement the parity test by checking whether $(d_{G^2}A)_{ij} \geq \deg(j)d_{G^2}(i,j)$. If this is the case, then we know $d(i,j)$ is even; if it's not, then it's odd.

This extra step takes n^ω time (but now n^ω is integer multiplication). And again you get your log factor correspondign to dropping the diameter.

Ont eh pset, we'll have to think about how to implement the parity test without using integer MM, just Boolean. (It is possible, and we're given a hint.)

§3 February 11, 2025

Today we'll continue talking about algorithms for APSP using fast matrix multiplication.

§3.1 APSP

As a recap of what we know, we know that APSP when weights are unbounded integers has an algorithm running in $n^3/\exp(\sqrt{\log n})$ time, by Williams (2014). Last time we showed that if your graph is undirected and unweigthed, Seidel showed you can do $\widetilde{O}(n^\omega)$; this is conditionally optimal because undirected unweighted APSP solves Boolean multiplication, which we believe is n^ω time. This algorithm was extended to work for weights in $\{1,\ldots,M\}$, with cost $\widetilde{O}(Mn^\omega)$ (this is pseudopolynomial). If $\omega = 2$, you can get subcubic runtimes for weights sublinear in n. This is by Shoshan and Zwick (it's somewhat complicated, but there have been simplifications later on).

Today we'll look at directed graphs. We won't be able to get n^ω (and there are reasons to believe you can't); but we'll get something better than cubic when weights are bounded in some interval $\{-M,\ldots,M\}$. (We can even handle negative weights, as long as there are no negative cycles. Negative weights for undirected graphs are a separate can of worms because if you can reach a negative-weight edge, you can go back and forth — it basically forms a negative cycle.)

(Recall that $\widetilde{O}$ kills polylog factors; we won't care about them, but there will never be more than 3 or 4.)

First, we'll get an $\widetilde{O}(\sqrt{M} \cdot n^{(3+\omega)/2})$ algorithm, due to Alon, Galil, and Margalit from the 1990s. The next is an improvement if $\omega > 2$ — it'll get $\widetilde{O}(M^{1/(4-\omega)}n^{2+1/(4-\omega)})$, and it's by Zwick. If $\omega > 2$, then $1/(4-\omega) > 1/2$, and you can check this algorithm is faster than the other. But if $\omega = 2$, then both have the same runtime – both are $\widetilde{O}(\sqrt{M}n^{2.5})$.

This is not the best expression for Zwick's runtime; you can do a bit better using the best-known bound on *rectangular* multiplication — but we won't do that, since we don't have nice expressions and you have to run some optimization to figure out the best exponents.

Why is this 2.5 and not 2? In Seidel's algorithm, we used the triangle inequality in both directions. The fact you can't do this in directed graphs is really what stops us.

§3.2 The short path-long path framework

We'll see the framework for both these algorithms — AGM is the base, and Zwick improves one of the steps.

Yoru input is some directed graph $G = (V, E)$, with some weights $w\colon E \to \{-M, \dots, M\}$. And there's no negative cycles.

First, we'll remember that we can compute the transitive closure of G in $\widetilde{O}(n^\omega)$ — this tells us, for every pair of vertices, whether there *exists* a path between them. For pairs (u, v) where there isn't a path, we immediately know their distance is $+\infty$, and we can disregard them. So for the rest, without loss of generality, we'll only consider pairs (u, v) which do have a path between them.

For the others, we'll pick *some* shortest path P_{uv}. Every pair of vertices has some path between them, so we pick some shortest path. (Because the graph has weights, these paths could have different number of edges.) For the sake of argument, we're going to pick just one path and only discuss that path.

For that path, we define $h(u, v)$ as the number of vertices on P_{uv} (also called *hops*). We also define $d(u, v)$ as the weight of P_{uv}. (If the graph is unweighted these are the same, up to ± 1's, but if it's weighted they can be different.)

Both algorithms do two steps. They consider pairs where the number of hops on their shortest path is small, and pairs where the number of hops is large, and they handle the two different hop-lengths differently.

So we pick some parameter K. We'll first compute, for every (u, v),

$$d_{<K} \begin{cases} = d(u,v) & \text{if } h(u,v) < K \\ \geq d(u,v) & \text{otherwise.} \end{cases}$$

So this computes upper bounds on the distances such that they're correct whenever the hop-length is short. And we'll also compute

$$d_{\geq K} \begin{cases} = d(u,v) & \text{if } h(u,v) \geq K \\ \geq d(u,v) & \text{otherwise.} \end{cases}$$

Today we'll do this with high probability (i.e., with probability at least $1 - 1/\mathsf{poly}(n)$, this holds for every (u, v)); later on we'll see how to derandomize it.

Finally, once we have these two values for every pair of vertices, how do we get the distances? We just take the minimum — the hop-length is in one of these two cases, and since they're always *at least* the distance, when you take the minimum you get the correct answer. So for all (u, v), we have

$$d(u, v) = \min\{d_{<K}(u, v), d_{\geq K}(u, v)\}.$$

So that's all we have to do.

These two algorithms (AGM and Zwick) compute $d_{\geq K}$ in exactly the same way. So we'll do that first. And where they differ is how they handle $d_{<K}$.

§3.3 Long paths

Now we'll handle the second case, where the hop-lengths are at least K. So we consider two vertices u and v, with some path P_{uv} between them; and we'll assume their hop-length is at least K.

A general phenomenon in algorithms is that when something is large, you can randomly sample and hit it. There's this general hitting set lemma that we're going to state and maybe prove later on, and we're going to use it to solve this problem.

Lemma 3.1

Suppose we have subsets $S_1, \ldots, S_N \subseteq \{1, \ldots, n\}$, where $N = \mathsf{poly}(n)$, such that for every i, we have $|S_i| \geq K$. Then if $S \subseteq \{1, \ldots, n\}$ is uniformly random of size $|S| \geq \Theta((n \log n)/K)$, then with high probability, for every i, $S_i \cap S \neq \emptyset$.

This says: Suppose you're given a bunch of subsets from a universe, where the number of subsets is polynomial. For us, $\{1, \ldots, n\}$ are the vertices, and the S_i's are the vertex sets of paths with hop length at least K. So if S_i is a path with at least K nodes, S_i is all the nodes of the path, so $|S_i| \geq K$. So if I pick a uniformly random set S of size $n/K \cdot \log n$, then it contains an element from every one of our sets S_i with high probability. And the larger the sets S_i are, the smaller this sample S needs to be. (This is not hard to prove; you can try, and we might prove it later.)

In our setting, we have these paths; there's at most n^2 paths of length P_{uv}, which is this N. So every path gives us a set of vertices (the set of vertices on the path), which is a subset of $\{1, \ldots, n\}$; and the number of vertices is at least K.

Then if we pick a random subset S of size $|S| \geq \Theta(\frac{n}{K} \log n)$, then with high probability, for every P_{uv} with hop length $h(u, v) \geq K$, $S \cap P_{uv} \neq \emptyset$.

So that means even though I don't know these paths P_{uv}, if I pick a random subset of vertices, this subset will contain at least one node on every one of these long paths (with high probability).

Now what I'm going to do is I'm going to run single-source shortest paths from every vertex in S. So we have our set S, and every uv that has a long shortest path passes through it. So if I run SSSP from every vertex in S in the out-direction and the in-direction (since the graph is directed), I can combine them to get an upper bound for the distance between u and v.

So for every $s \in S$, we run $\mathsf{SSSP}_{\text{in}}(s)$ and $\mathsf{SSSP}_{\text{out}}(s)$. And then for every $u, v \in V$, we set

$$d_{\geq K}(u, v) = \min_{s \in S} d(u, s) + d(s, v).$$

We'll get to the runtime in a second, but first let's look at correctness. For every pair of vertices, we don't know the hop-length. But if the hop-length happens to be at least K, then we know one of these little s's is correct. And we have these two distances, so when we take the minimum over all s's, we get $d(u, s) + d(s, v) = d(u, v)$ (because we have hit a vertex on the shortest path). So we definitely have $d(u, v)$ when $h(u, v) \geq K$.

And why is the second part true — for all the other vertex pairs, our distance estimate is *at least* the distance? Everything we're minimizing over is some path (or ∞) — the path from u to s to v — so it's at least the length of the shortest path. (In other words, $d(u, s) + d(s, v)$ is always either ∞ or the weight of som epath, so it's at least $d(u, v)$; and if s is on the shortest path, it's exactly correct.)

So we get the large-hop case.

Now let's look at the running time. The second part (taking a min over s) is just $n^2 |S| = \widetilde{O}(n^3/K)$ (since we're taking a min over $|S|$ things); this is going to be smaller than the first part.

And for the first part, we're running $\Theta(\frac{n}{K}\log n)$ SSSP computations. If weights are nonnegative, you can run Djikstra's algorithm, which has runtime at most n^2. If weights are negative, however, you might wonder what happens. Fortunately int eh last year or two, there have been papers showing that even in negative-weight directed graphs, you can solve SSSP in $\widetilde{O}(n^2)$ time. But even before we knew that, we could have solved this problem in the same runtime. The reason for that is when there's negative weights, it suffices to run a *single* SSSP computation. If you've taken undergraduate algorithms, you may have seen Johnson's trick, which says you can compute SSSP from some arbitrary node, and then these distances can serve as *potentials* — and you can use them to reweight the edge weights in the whole graph so that now they're nonnegative. So a single SSSP computation suffices to reweight your graph; and then every new SSSP computation can be done with Djikstra's. So the rest still take $\widetilde{O}(n^2)$ time. And you can do a single SSSP computation in $O(Mn^\omega)$ time, and that suffices. But now we can just say that you can do SSSP in $\widetilde{O}(n^2)$. So the point is that now this also becomes $\widetilde{O}(n^3/K)$.

So $\widetilde{O}(n^3/K)$ suffices for both parts — to solve SSSP and collect the answers in the end. And that solves the second case (of computing $d_{\geq K}$).

Student Question. *How do you handle the randomness — how do you know if you're in a universe where S is not good?*

Answer. Here you can't — you don't know if you've computed correctly. But later on we'll see you actually don't need the randomness. Why not? You have some relaly long path, but somehow you've already computed the short paths — the paths at most K. So you already have the first part (from u into S). And once you have that, you already know it. So you don't need randomness — all you need is a greedy hitting set procedure that will hit this short path. And then you already have a node on the first piece of the short paths, and you can deterministically solve the problem. But we'll see that later; for now we'll just use the hitting set lemma and not worry about it.

Student Question. *Even if you use Johnson's algorithm for SSSP, isn't the first Bellman–Ford step still going to take $O(n^3)$ time if the graph is dense?*

Answer. That's why the first SSSP algorithm is done with something like $\widetilde{O}(Mn^\omega)$ — there is a way to run SSSP in this time (in directed weighted graphs), which we'll see next time. But nowadays there are algorithms that run in almost linear time, even without multiplication, so we can avoid it.

So we'll stop thinking about long paths, since they're easier in some sense; and now we'll focus on short paths, which seem to be the hardest ones.

§3.4 Short paths — AGM

Now we'll move on to short paths. We'll first do the simple way, which is the AGM algorithm.

In the long paths case, we didn't use matrix multiplication; that can be done pretty combinatorially. But here we will use it.

Definition 3.2. Given a graph G, its generalized adjacency matrix is defined by

$$A_{ij} = \begin{cases} 0 & \text{if } i = j \\ w(i,j) & \text{if } (i,j) \in E \\ \infty & \text{otherwise.} \end{cases}$$

So A_{ij} is the minimum weight of a path from i to j of hop-length at most 1.

Now we'll define a matrix product, which has many names (the $(\min,+)$ product or *distance* product or the 'product over the tropical semiring' and so on).

Definition 3.3. Given two matrices A and B (with integer entries), their $(\min, +)$ product is

$$C_{ij} = \min_k (A_{ik} + B_{kj}).$$

We often write $C = A * B$.

SImilarly to the normal matrix product, this is associative; so we can compute powers of the generalized adjacency matrix under the $(\min, +)$ product. If we consider A^2, we have

$$A^2_{ij} = \text{min weight of an } i \to j \text{ path of hop length} \leq 2.$$

Claim 3.4 — We have A^k_{ij} is the minimum weight of an $i \to j$ path of hop length at most k.

If k happens to be a power of 2 (which we can assume is the case), then we can use successive squaring to compute

$$A^{2^i} = A^{2^{i-1}} * A^{2^{i-1}}$$

(very similar to how we argued about transitive closure under the normal matrix product).

Something that's not good for us is that the $(\min, +)$ product itself is *equivalent* to APSP, when the weights are unbounded — the best-knowon algorithm we know for this problem is just a bit better than n^3. What's happening here is that even though in our case A has weights in $\{-M, \ldots, M\} \cup \{\infty\}$, how big are the weights in A^{2^i}? The weights in $A2^{i-1}$ can go all the way to $M \cdot 2^{i-1}$, so they kind of blow up. The reason is this captures paths of length at most 2^{i-1}, and each edge of this path could have weight M.

So the weights grow. This is why when you multiply these matrices in the $(\min, +)$-product, these $(\min, +)$-products become more expensive. And we want to quantify how expensive it gets.

Claim 3.5 — If we have $n \times n$ matrices A and B with entries in $\{-M, \ldots, M\} \cup \{\infty\}$, then we can $(\min, +)$-multiply them in time $\widetilde{O}(Mn^\omega)$.

This is the best-known way to multiply. So basically the larger the weights are, the more you pay in front of n^ω.

Assuming this claim, what's the runtime for our successive squaring algorithm? In the step for 2^{i-1}, the runtime is $\widetilde{O}(M2^{i-1}n^\omega)$. And when we sum them up, we keep going from $i = 0$ to $\log_2 K$, so we get $\widetilde{O}(MKn^\omega)$. Now, MKn^ω is what we'll pay if we do it this way.

§3.4.1 Runtime of AGM

We can now combine it with the $\widetilde{O}(n^3/K)$ algorithm for the other case. We'll prove this claim in a little bit. But for now, suppose we have an $\widetilde{O}(MKn^\omega)$ algorithm for short paths, and an $\widetilde{O}(n^3/K)$ for long paths. In order to get our final algorihtm, the runtime will be

$$MKn^\omega + \frac{n^3}{K}$$

(there's some n^2 overhead, but it's dominated by the other things). We get to pick K, so we should choose it as the geometric mean of Mn^ω and n^3 — this is because when you have a runtime which is the sum of two terms, you should set them equal to figure out where it's minimized. So we set these equal and get $K^2 = n^{3-\omega}/M$, which means $K = n^{(3-\omega)/2}/\sqrt{M}$, and our fianl runtime is $n^{(3+\omega)/2}\sqrt{M}$, which is what we promised.

§3.4.2 Proof of Claim 3.5

Now we have to prove the claim. First, it would be nice to work with matrices that don't have ∞'s. So how do we get rid of ∞'s? We have A and B, and we want to compute $A * B$; and hteese guys have entries in $\{-M, \ldots, M\} \cup \{\infty\}$, and we want to get rid of the ∞'s. How do we do that? We can just make ∞ into $3M + 1$.

Why $3M$? We have

$$(A * B)_{ij} = \min_k (A_{ik} + B_{kj}).$$

Any *true* sum here, which doesn't include an ∞, is between $-2M$ and $2M$. If you make ∞ into $3M + 1$ (e.g., for some term B_{kj}), then even if A_{ik} were $-M$, you'll have $3M + 1 - M = 2M + 1 > 2M$.

So you can take the ∞ weights and replace them with $3M+1$'s, and once you compute the $(\min, +)$-product with the new entries, you'll be able to know when the correct answer is ∞ or not (it'll be ∞ when your number is at least $2M + 1$, and finite otherwise). And the nice thing is this only blows up the entries by a constant factor (e.g., a factor of 4).

So now we replace M with $3M + 1$ (this only costs a constant factor, which gets absorbed into the $O(\bullet)$).

So from now on, we only need to consider matrices with entries in $\{-M, \ldots, M\}$ when proving this claim; then we'll automatically also get it for ones with ∞'s, by this replacement.

So now let's do that.

So now we have these A's and B's with entries in $-M$ and M. And the first thing we're going to do is create new matrices out of them. We define A' as

$$A'_{ij} = M - A_{ij} \in \{0, \ldots, 2M\}.$$

Similarly, we define B' as

$$B'_{ij} = M - B_{ij} \in \{0, \ldots, 2M\}.$$

Now instead of computing the $(\min, +)$-product, we'll try to compute the $(\max, +)$-product defined by

$$C'_{ij} = \max_k (A'_{ik} + B'_{kj}).$$

(These entries are all nonnegative.) Why are we doing this? The $(\max, +)$-product has an easy translation into normal integer matrix multiplication! WHat we'll do is compute the normal (integer) matrix product of A'' and B'', where

$$A''_{ik} = (n+1)^{A'_{ik}} \quad \text{and} \quad B'_{kj} = (n+1)^{B'_{kj}}.$$

Why? We have

$$(A''B'')_{ij} = \sum_{k=1}^{n} (n+1)^{A'_{ik} + B'_{kj}}.$$

And nicely, $A'_{ik} + B'_{kj}$ appears here (this is what we wanted in the $(\max, +)$ product). Recall that the $(\max, +)$-product is defined as

$$C'_{ij} = \max_k A'_{ik} + B'_{kj}.$$

Now, for every (i, j), C'_{ij} appears somewhere here, in one of the exponents (the maximum term); this means

$$(A'' \cdot B'')_{ij} \geq (n+1)^{C'_{ij}},$$

because one of these exponents is C'_{ij}, and this is a sum of nonnegative (actually positive) terms.

And because every one of these terms is at most $(n+1)C'_{ij}$, the sum is also at most

$$n \cdot (n+1)^{C'_{ij}} < (n+1)^{C'_{ij}+1}.$$

And notably, this menas $(A''B'')_{ij}$ will give us C_{ij} — we can find the largest c such that this quantity is between $(n+1)^c$ and $(n+1)^{c+1}$. So once we've computed this matrix product, we can extract (or read off) C'_{ij} from it.

Then the question is, how long does it take us to compute hte actual matrix product $(A'' \cdot B'')_{ij}$, of these two matrices whose entries are enormous?

Last time Virginia mentioned that when we care about the sizes of entries, the matrix product of two matrices whose entries are integers with b bits can be computed in bn^ω time. So $n \times n$ matrix multiplication with b-bit entries is in time $\widetilde{O}(bn^\omega)$. In the homewor, there's some problem to show this for b^2n^ω; but you can actually do it with $b \log b$ using FFT, and the $\log b$ goes into the $\widetilde{O}$.

And what is the bit complexity of our guys A'' and B''? Here we have $\underline{b} = \log(n+1)^M = O(M \log n)$. So for us, computing $A'' \cdot B''$ will take $\widetilde{O}(Mn^\omega)$ (the log disappears in the $\widetilde{O}$). And this is how you get $\widetilde{O}(Mn^\omega)$ time.

So you take $(\min, +)$ product and convrt it to $(\max, +)$ product with nonnegative entries; this naturally embeds as a normal matrix product with enormous entries, and you pay a factor for those enormous entries.

§3.5 Zwick's algorithm

What we've done so far is gotten the AGM algoirhtm with two tools — hitting sets with random sampling to handle long paths, and the $(\min, +)$-product with small entries to handle short paths. Now we'l see how to get Zwick's runtime. The way this will work is it'll take the same appraoch for long paths. For short paths, it'll also use this claim; but you can actually use the hitting set algorihtm *inside* the short paths as well. (Here we didn't; we just used successive squaring.)

In successive squaring, we were taking a $n \times n$ matrix corresponding to hops of length at most 2^{i-1}, and multiplying it by itself. However, these matrices $A^{2^{i-1}}$ have shape $n \times n$. But we can actually save in their dimensions!

§3.5.1 Intuition

Here's some intuition, which we'll later make formal. Suppose I have a path from u to v, of length i. Before what we were doing is a $2^{i-1} \times 2^{i-1}$ sort of piece. But we already know via the hitting set argument that you can sample some set S of size $\frac{n}{2^i} \log n$, and you will be able to hit this path. And therefore, instead of doing a $(\min, +)$ product of $n \times n$ by $n \times n$, you only really need the columns on the left and rows on the right that correspond to the sample S — because here you have $A^{2^{i-1}} \times A^{2^{i-1}}$, but because you know already you're looking for paths of length 2^i, you can find them with a smaller set. Because from the long-path framework, if we can get distances from u to s and s to v, then we can minimize over S.

So this lets us save over computation — instead of having $n \times n$ by $n \times n$, the middle will be $n/2^i$.

If we only had an algorithm for *square* matrix multiplication, we could still solve this rectangular matrix multiplication using squares. How? You can divide it into square subsections — let's take this very long rectangular thing and make it into squares, of size $n/2^i \times n/2^i$. There's 2^i squares in each of our two matrices, and the total number of products is 2^i. So I can use the square multiplication algorithm to compute the rectangular one, and it'll be cheaper.

So this is the intuition — the longer the paths get, the smaller the sample I need. And the thing is, though, we only have this algorithm whose runtime depends on the weights; the weight of this path will become like $M \cdot 2^i$, so we'll actually have to pay this in front. But you're saving by the fact that this gets smaller.

We'll have to make this formal; it's not formal yet, and in fact to make it formal, this 2 will have to change to a 3/2 (and we'll see why).

§3.5.2 Formalization

Here's what we're going to do. We'll first say, when the paths have a *very* small number of hops (e.g., 2), we just do our $(\min,+)$-product of the adjacency matrix with itself. At some point, the number of hops will grow. And at that point, we want to reduce the sample.

Let's suppose we have computed some distance estimates $\widetilde{d}(i,j)$ such that

$$\widetilde{d}_\ell(i,j)\begin{cases}=d(i,j) & \text{if } h(i,j)\le (3/2)^\ell\\ \ge d(i,j) & \text{otherwise}\end{cases}$$

(for some ℓ; we'll explain later why we have 3/2). Now we want to compute

$$\widetilde{d}_{\ell+1}(i,j)=\begin{cases}=d(i,j) & \text{if } h(i,j)\le (3/2)^{\ell+1}\\ \ge d(i,j) & \text{otherwise.}\end{cases}$$

So suppose we're given this $\widetilde{d}_\ell(i,j)$. Now let's consider some u and v such tha $th(u,v)\le (3/2)^{\ell+1}$. We want to compute $\widetilde{d}_{\ell+1}$ for it.

There's two cases. First, if $h(u,v)\le (3/2)^\ell$, then we're already done — because we already have an estimate for its distance, from $\widetilde{d}_\ell$. So

$$d_{\ell+1}(u,v)=d_\ell(u,v),$$

which we've already computed. Otherwise, we have $h(u,v)\in((3/2)^\ell,(3/2)^{\ell+1}]$. So we're in this interval.

Now let's look at u and v, and the shortest path P_{uv}. Now we're going to split it into three equal parts — the first part, the second part, and the third part, each of length roughly $h(u,v)/3$ (with some ±1).

Why do we do this? The reason we split into three parts i sbecause first, the number of hops in the first two segments is

$$2h(u,v)/3\le 2/3\cdot(3/2)^{\ell+1}=(3/2)^\ell.$$

So this part is at most $(3/2)^\ell$. But also, the second and third parts combined is also at most $(3/2)^\ell$. So in aprticular, for *every* node in the middle, we actually know the distance from u to it and from it to v. So for every x in the middle third, $d_\ell(u,x)=d(u,x)$ and $d_\ell(x,v)=d(x,v)$. So for every node in the middle, no matter which one, we have the correct distance already computed for both parts.

And this means if I can somehow guarantee that I have found some node on this path, I will be able to get the exaxct distance by taking $d(u,x)+d(x,v)$!

Another property of this is that the length of the middle third is big — because $h(u,v)\ge(3/2)^\ell$. So the number of nodes on the middle third is

$$\frac{h(u,v)}{3}\ge\frac{(3/2)^\ell}{3}.$$

And for every fixed ℓ, I can pick a random sample of the vertices of a certain size to guarantee that for every one of these paths whose hop length is in $[(3/2)^\ell,(3/2)^{\ell+1}]$, we'll have a node in the middle third.

So we take $S\subseteq V$ to be a random set of size

$$|S|\sim\frac{n}{(3/2)^\ell}\log n.$$

Then with high probability, for every (u,v) with $h(u,v)\in((3/2)^\ell,(3/2)^{\ell+1}]$, S contains some node in the middle third of P_{uv}.

So now we have this set S of this size. And how do I get the distance $d_{\ell+1}(u,v)$ from it? We set

$$d_{\ell+1}(u,v)=\min\left\{d_\ell(u,v),\min_{s\in S}d_\ell(u,s)+d_\ell(s,v)\right\}$$

(the first $d_\ell(u,v)$ term is in order to handle the case $h(u,v) \leq (3/2)^\ell$). And this second term is just a $(\min,+)$-product, which looks like this: On the left, we have the subjatrix of d_ℓ defined on $V \times S$, so we have n rows (corresponding to all the vertices), and the columns correspond to just the vertices in S, so there's $\widetilde{O}(n/(3/2)^\ell)$ of them. And then we $(\min,+)$-multiply it by basically the same thing, sideways — if we call the first thing $d_\ell(*,S)$, then this second thing is $d_\ell(S,*)$.

How large are the entries here? They're in $\{-M(3/2)^\ell,\ldots,M(3/2)^\ell\} \cup \{\infty\}$ (the reason for that is that this captures all the weights of paths whose hop-lengths are at most $(3/2)^\ell$, so the number of edges on these paths is this many, and each edge has weight between $-M$ and M; and you have some ∞'s for when the distances are undefined).

And so by our claim Claim 3.5, the runtime to compute these things if they happened to be square would be $M(3/2)^\ell \cdot n^\omega$. But they're *not* square! Instead, we do what we proposed earlier — we break them into pieces of size $|S|$ and compute the product. So we have pieces d_1, …, $d_{(3/2)^\ell}$ (where $(3/2)^\ell$ comes from $n/|S|$). And we multiply every block by every other block; so the number of $|S| \times |S|$ products we need to compute is

$$\left(\frac{n}{|S|}\right)^2 = \widetilde{O}((3/2)^{2\ell}).$$

And each $|S| \times |S|$ $(\min,+)$-product computation costs us

$$M\left(\frac{3}{2}\right)^\ell \cdot |S|^\omega .$$

So the total cost (ignoring polylogs) is

$$\left(\frac{3}{2}\right)^{2\ell} \cdot M\left(\frac{3}{2}\right)^\ell \cdot \left(\frac{n}{(3/2)^\ell}\right)^\omega = Mn^\omega \cdot \left(\left(\frac{3}{2}\right)^\ell\right)^{3-\omega}.$$

Since $3-\omega > 0$, this $(3/2)^\ell$ will be biggest when ℓ is biggest — and we're going over growing ℓ, where $\ell = 0$, 1, …, $\log_{3/2} K$ (where K is our bound on what a short-length path means). So this is always at most

$$Mn^\omega K^{3-\omega}.$$

Now this is our final runtime for the short path-lengths.

We want to contrast this with our previous bound of $\widetilde{O}(MKn^\omega)$. Here K became $K^{3-\omega}$; if $\omega > 2$, then $3-\omega < 1$, so this is strictly beetter.

This means if we replace the short-paths step with this one, we'll do better if $\omega > 2$.

§3.5.3 Final computations

Now we can put things together to get the runtime stated earlier. We had n^3/K for long paths, and $Mn^\omega K^{3-\omega}$ for short paths. So we set

$$K = \frac{n^{(3-\omega)/(4-\omega)}}{M^{1/(4-\omega)}},$$

and you get your runtime of $\widetilde{O}(M^{1/(4-\omega)}n^{2+1/(4-\omega)})$.

Note that if $\omega \to 2$, then this exponent goes to $n^{2.5}$.

This is Zwick's algorithm, and it's the best we know. (The only improvement is in the blocking part — we blocked into square pieces, but you can use the best-known algorithm to multiply skinny rectangular matrices, and you get something a bit better that works out to $n^{2.53}$.)

§4 February 13, 2025 — Yuster–Zwick Distance Oracle

Today we'll continue what we started last time — last time we talked about directed APSP with bounded integer weights. Today we'll continue doing this, but instead of solving APSP, we'll build a *distance oracle* — a data structure that preprocesses the graph and stores some information so that distance queries can be answered fast. It turns out that this can be computed faster than actually precomputing all the distances.

§4.1 Setup for distance oracles

We'll see distance oracles throughout the coure later as well. A *distance oracle* (DO) is a data structure: Given a graph $G = (V, E)$ (which can be directed or undirected, weighted or unweighted), you get to preprocess it (the *preprocessing time* matters). And after preprocessing, you store some stuff (here, the *storage space* matters). And then you need to be able to answer queries. Here, usually you're given some query, which is a pair of vertices (u, v), and you want to answer by outputting an estimate of the distance — you output some $\widetilde{d}(u, v)$, which is hopefully close to the distance you want. So here, we want this to be a good estimate of the actual distance $d(u, v)$. 'Good' could mean *exact*, or it could mean there's an additive error, or a multiplicative error, or both; we'll see multiple examples of all of this. So the *approximation quality* matters here; and we also care about the *query time*.

§4.2 Main theorem

What we're going to do today is give a distance oracle for directed graphs with bounded integer weights which has better preprocessing time than APSP, okay storage space, excellent approximation quality (it'll be exact), and okay but not great query time.

Theorem 4.1 (Yuster–Zwick 2005)
There is an $\widetilde{O}(Mn^\omega)$ time algorithm (for preprocessing) such that for every directed $G = (V, E)$ with weights in $\{-M, \dots, M\}$ and no negative cycles, it computes an $n \times n$ matrix $\widetilde{D}$ such that with high probability, for every pair of vertices $u, v \in V$, we have

$$d(u, v) = \min_{x \in V} \widetilde{D}(u, x) + \widetilde{D}(x, v).$$

(This is 'with high probability,' but it can be derandomized.)

So this $n \times n$ matrix is our storage; it'll have $n^2 \log(Mn)$ bits.

And then min means the query time is $O(n)$, since you can try all the n vertices and check this quantity; and it'll compute the exact distance.

The approximation quality is 1 (this is exact). The storage space is not exact — you have to store the adjacency matrix anyways, since if you want an exact oracle you need to do reachability; and $\log Mn$ is an extra part encoding how long the distances can be (the largest weight of a path could be Mn). So the storage space is pretty good. The preprocessing time Mn^ω is the same runtime as for *undirected* APSP — recall that for directed APSP we had a much worse algorithm — so we are beating that. But it's still not great — we're still using matrix multiplication. (You can't really avoid it, because it's an exact algorithm, but still.) But the query time is not great — you can't *compute* all the distances (then you'd have to spend $n^2 \cdot n = n^3$ time to extract them from the oracle). But the oracle would be really great if you only wanted a small number of distances, e.g., if you wanted to do a small number of SSSPs.

Corollary 4.2

SSSP can be done in $\widetilde{O}(Mn^\omega)$ time on directed graphs.

What we do is compute the oracle; and then we have some source from which we want to compute the distances; and then you must look up the n distances $d(s,v)$. Each of the query times is n, so we get $n^2 + Mn^\omega$.

You can even do better — you can do any small number of completely unrelated pairs, which is a potentially harder problem, in this much time as well.

You can also detect negative cycles in this time (this is a corollary of the algorithm, not the statement — you can use the algorithm in there to detect negative cycles). You can also compute the shortest cycle in the graph in the same time (this takes some thought).

Student Question. *Are we saying d is $(\min,+)$ of $\widetilde{D}$ and itself? Couldn't you get that with one matrix multiplication?*

Answer. The problem is that the entries in $\widetilde{D}$ are really big — they're between $-Mn$ and Mn — so doing the $(\min,+)$-product here would be way too expensive.

What we're going to do here is not going to remember we assume everything from last lecture, so we'll go through the tools and recap what we did for Zwick's algorithm (which this is based on); and then we'll see how to change it to get this.

§4.3 Tools from last time

Let's review some tools from last time.

The first one is that you can use square matrix multiplication to multiply rectangular matrices: Suppose $m < n, k$, and you're given two matrices A (which is $k \times m$) and B (which is $m \times n$), where m is the smaller dimension. Then we can split A into $m \times m$ blocks $A_1, \ldots, A_{k/m}$ and B into $m \times m$ blocks $B_1, \ldots, B_{n/m}$. And then their product is the $k \times m$ matrix

$$\begin{bmatrix} A_1B_1 & A_1B_2 & \cdots \\ A_2B_1 & \cdots & \ddots \end{bmatrix}$$

(where each entry is A_iB_j). So to compute this, you just compute each pair of products of square matrices. Then the runtime is

$$(k/m) \cdot (n/m) \cdot m^\omega = \widetilde{O}(knm^{\omega-2}).$$

If each of these has b-bit entries, then you'd just multiply here by b.

So this is the frist simple tool.

The second tool is that a $(\min,+)$-product of $k \times m$ by $m \times n$ matrices (where $m \leq k \leq n$) with entries in $\{-M, \ldots, M\} \cup \{\infty\}$ can be reduced to integer matrix multiplication of $k \times m$ and $m \times n$ matrices whose entries have $M \log n$ bits. And now we can combine this with the first tool; so this rectangular $(\min,+)$ product can be computed in

$$\widetilde{O}(Mknm^{\omega-2})$$

time (where $\widetilde{O}$ hides log factors).

The reason we do this is for example if $k = n$ and $m < n$, then the runtime becomes $Mn^2m^{\omega-2} < Mn^\omega$. So instead of doing $n \times n$ matrix multiplication we'll sometimes do $n \times m$ and $m \times n$, and when $m < n$ we'll get a smaller runtime. (Today we'll also have different values of k than just $k = n$, as last time. And the smallest dimension measures how much we save.)

Now we'll recap a few more tools, and then we'll get to the meat. The next thing is the hiting set lemma:

Lemma 4.3 (Hitting set lemma)

Let $S_1, \ldots, S_N \subseteq \{1, \ldots, n\}$ where $N = \mathsf{poly}(n)$ and $|S_i| \geq K$ for all i, and suppose we choose $S \subseteq \{1, \ldots, n\}$ with $|S| \sim \Theta(\frac{n}{K} \log n)$ uniformly at random. Then with high probability, $S \cap S_i \neq \emptyset$ for all i.

So if we have a large enough random sample of the universe and your original sets are big enough, then your random sample will hit all of them with high probability. This is useful when the S_i are unknown.

We used this in Zwick's algorithm as follows. Here P_{uv} is some shortest path from u to v. We don't know that shortest path; we want to compute it. So it's unknown in the same sense as the sets S_i above. We'll use $h(u, v)$ to denote the length of this path (either the number of nodes or edges; it doesn't really matter, since they're the same up to 1), which we call the number of *hops*; and $d(u, v) = w(P_{uv})$.

In Zwick's algorithm, we looked at a path where $h(u, v) \in [(3/2)^i, (3/2)^{i+1}]$, and we defined the 'middle third' of the path; this had $h(u, v)/3$ nodes.

We also had a sample S_i, which was a random sample from V of size

$$|S_i| \sim \frac{n}{(3/2)^i} \log n.$$

And we had that for every such (u, v), this sample would hit some node in the middle third — i.e., S_i hits the middle third. We picked the middle third such that for every node in the middle third, the number of hops to both u and v is bounded — for every s in the middle third,

$$h(u, s), h(s, v) \leq \left(\frac{3}{2}\right)^i$$

(the nuber of hops from u to any node in the middle is at most the number of hops from u to the end of the interval, which is at most 2/3 of the number of hops on the entire path).

And then we used this to build an algorithm that could compute all the distances of paths with a small number of hops.

§4.4 Review of Zwick's algorithm

This is basically all we did last time. Now we're going to write down the steps of Zwick's algorithm. We won't write down the part that handles long paths; we're actually not going to need that this time. There we took a random sample and did SSSP in and out of every node in the sample, and then we cmobined these distances to figure out the distances on paths whose hop lengths were very long. But today we will not actually use that.

The part of Zwick's algorithm we *will* use is for computing $d(u, v)$ when $h(u, v) \leq K$ (where K is some parameter — we'll say $K = (3/2)^k$, or $k = \log_{3/2} K$).

We'll first write down how it worked, and then look at how we can improve it.

What did we do? For $\ell = 0, \ldots, k = \log_{3/2} K$, we were computing some distances. So we said S_ℓ was this random sample of size

$$|S| \approx \frac{n}{(3/2)^\ell} \log n.$$

And we had some initial distance matrix d_0, which is just the adjacency matrix A; this says all the distances for paths with hop lengths *at most* 1. More generally, we'll compute d_ℓ, which is the distances for (u, v) of hop length at most $(3/2)^\ell$; this is what we want to compute. And we assume we've computed $d_{\ell-1}$; and now we're going to get d_ℓ.

For this, we said that

$$d_\ell(u,v) = \min\{d_{\ell-1}(u,v), d_{\ell-1}(u,S_\ell) * d_{\ell-1}(S_\ell,v)\}.$$

What does this mean? If the hop length is at most $(3/2)^{\ell-1}$, then it was already computed in $d_{\ell-1}$. Otherwise we're in this above case, where we're hitting themiddle third with S_ℓ; and then you're taking the minimum of $d(u,s)$ and $d(s,v)$, both of which have at most $(3/2)^{\ell-1}$ hops.

And then in the end, you kind of output d_K.

There's one little caveat we didn't mention last time — when you do this computation with the $(\min,+)$-product, any distance bigger than $M\cdot(3/2)^{\ell-1}$ should be set to ∞. So you have to cap it. (This is because we only care about distances between u and something in the middle, and they'll never be bigger than this.) Once you cap it, we get to compute this $(\min,+)$-product fast. The computation of this $(\min,+)$ product is what determines the runtime of the algorithm. In particular, the runtime ends up being

$$M\cdot(3/2)^\ell\cdot n^2\cdot|S_\ell|^{\omega-2} = M\cdot(3/2)^\ell\cdot n^2\cdot\left(\frac{n}{(3/2)^\ell}\right)^\omega = Mn^\omega\left((3/2)^\ell\right)^{3-\omega}$$

(using the $Mn^2m^{\omega-2}$ from earlier — the n corresponds to the choices for u and v). And because $3-\omega>1$, this thing is maximized when $3/2$ is maximized, which is when it's K; so we get $Mn^\omega K^\omega$.

What we actually want to do today is make this $3-\omega$ appear — if we could get rid of it, making it be 0, then we'd just get Mn^ω.

And the way we're going to do that is by, instead of computing $(\min,+)$ products which are the vertex set, some sample, and then the vertex set itself, we'll compute the vertex set times a sample times *another* sample. So then one of these n's will become m, which will mean the 3 in $3-\omega$ becomes a 2, and you'll see it will disappear.

§4.5 The main idea

Here's what we're going to do. Instead of computing the distances for *every* pair of vertices, we're preparing some matrix that we'll store. So we'll actually compute the distances between a vertex u and something that hits this middle third (somehow), and something that hits this middle third and the other vertex v. So we don't actually want to compute this full $(\min,+)$ product of (v,S_ℓ) and (S_ℓ,v).

Here's our goal. Imagine that instead of taking fully random samples every time, we're going to *build up* a hierarchy of samples. So we'll start with the whole vertex set $V=S_0$. And then inside S_0, we will pick a completely random subset S_1. And then in S_1 we pick a completely random sample (of a certain size). And so on. SO we'll have

$$V = S_0 \supseteq S_1 \supseteq \cdots \supseteq S_{i-1} \supseteq S_i \supseteq \cdots \supseteq S_{\log_{3/2} n},$$

where S_i is a random sample of S_{i-1} of size roughly $n/(3/2)^i\cdot\log n$. (Ignore the $\log n$ for a moment.)

So in the beginning you have size n, then somewhere in the middle you have $n/(3/2)^i$, up until the bottom where you have roughly $\log n$ nodes.

Even though they're correlated, if I pick one of these guys in the middle, it's still a random sample of the original vertex set of the correct size. But having them nested like this will be really useful for our computations.

Now what I want to do is for every vertex in V, I want to compute the distance between v and every vertex in S_i, as long as the hop length between them is a certain size.

Goal 4.4. Compute two matrices d_i and $\overline{d_i}$ such that for $v \in V$ and $s \in S_i$, we have

$$d_i(v,s) \begin{cases} d(v,s) & \text{if } h(v,s) \leq (3/2)^i \\ \geq d(v,s) & \text{otherwise,} \end{cases}$$

and $\overline{d_i}(s,v)$ should be kind of similar —

$$\overline{d_i}(s,v) = \begin{cases} = d(s,v) & h(s,v) \leq (3/2)^i \\ \geq d(s,v) & \text{otherwise.} \end{cases}$$

So this is very similar to what we had before, except instead of trying to get things for *every* pair of vertices, we only look at pairs consisting of an arbitrary vertex and something in the sample S_i.

Now suppose we have computed d_{i-1}. First, what is d_0 (and $\overline{d_0}$)? Well, $S_0 = V$, so $d_0(V,V)$ has to have the correct distances for hop lengths at most 1. So d_0 is just hte generalized adjacency matrix A. (You need exact distances for all pairs of vertices as long as the hop length is at most 1; those are edges or just staying at the same vertex.)

Now you assume you have d_{i-1} and $\overline{d_{i-1}}$. And now we want to compute d_i. Where is d_i? You have some vertex v, and something in your sample $s \in S_i$, with a shortest path P_{vs} between them. And there's two choices for the hop-length — you could have $h(v,s) \leq (3/2)^{i-1}$, or you could have $h(v,s) \in [(3/2)^{i-1}, (3/2)^i]$ (or it could be bigger, but then we don't really care, and we can just set the distance to ∞).

The first case is where we use the fact that the subsets are *nested* — S_i is designed to be a subset of S_{i-1}. So in the first case, $d_i(v,s)$ can just be set to be $d_{I-1}(v,s)$ — because d_{i-1} contains, for every vertes and everything inside S_{i-1}, the exact distances when the hop lengths are at most $(3/2)^{i-1}$. And crucially we have $S_i \subseteq S_{i-1}$. So we can just look up what we've already computed (since we already have d_{i-1}); this contains everything for $v \in V$ and $s \in S_{i-1}$. And $s \in S_i \subseteq S_{i-1}$, so we can just look up th distance.

Otherwise, if we're in the case $h(v,s) \in [(3/2)^{i-1}, (3/2)^i]$, this is just like in Zwick's algorithm. So we look at the path from v to s and the middle third. Then S_{i-1} hits this middle third with high probability — this middle third is very large (of size $\Omega((3/2)^{i-1})$). So S_{i-1} hits this middle third with high probability (for every single pair (v,s) whose hop length is in here).

So we have a node $s' \in S_{i-1}$ in this middle third from v to s. Now in order to compute the distance, I can combine $d(v,s')$ and $d(s',s)$.

Now I'm goign to redraw this picture, and write it as a $(\min,+)$ product. We have v, and then $s \in S_i \subseteq S_{i-1}$, and we now hav e$h(v,s) \in ((3/2)^{i-1}, (3/2)^i]$. So we look at the path from v to s, and the middle third; and we can find $s' \in S_{i-1} \cap$ middle third. Then we have

$$d(v,s) = \min_{s' \in S_{i-1}} d(v,s') + d(s',s).$$

Now, for $d(v,s')$, because $s' \in S_{i-1}$ and becaus ethis hop-length $h(v,s') \leq (3/2)^{i-1}$ is small, we have that

$$d(v,s') = d_{i-1}(v,s').$$

Similarly, we have $h(s',s) \leq (3/2)^{i-1}$, and both s' and s are in S_{i-1} (in fact), so we can use either d_{i-1} or $\overline{d_{i-1}}$ (it doesn't actaully matter) to compute it. And then we get

$$d(v,s) = \min_{s' \in S_{i-1}} (d_{i-1}(v,s') + d_{i-1}(s,s')).$$

So we should set

$$d_i[V,S_i] = \min\{d_{i-1}[V,S_i], d_{i-1}[V,S_{i-1}] \star d_{i-1}[S_{i-1},S_i]\}$$

(where $d_i[V,S_i]$ denotes the whole matrix).

Student Question. *How do you choose* all *the S_i's to be nested?*

Answer. You start with $S_0 = V$ and then at each step, you take a random subset of the previous one. (You could also pick every vertex from S_i with the same probability to get the correct expected size, depending on what you wanted to achieve. If you wanted to do this deterministically, then you'd need to do some greedy thing, but then you'd need to know the paths. We're trying to avoid that right now. But we'll have another lecture where we discuss hitting sets.)

Symmetrically, we can say

$$\overline{d_i}[S_i, V] = \min\{\overline{d_{i-1}}[S_i, V], \overline{d_{i-1}}[S_{i,S_{i-1}}] \star \overline{d_{i-1}}[S_{i-1}, V]\}$$

(we have a directed graph, so we have to do both directions). So you do this.

§4.6 Runtime

Now let's look at the running time of this thing. The nice thing about $d_i$is that we have a $n \times |S_{i-1}|$ matrix on the left (for d_{i-1}), and we're multiplying by a $|S_{i-1}| \times |S_i|$ matrix. The sizes of S_i and S_{i-1} are the same, up to constants; they're $\widetilde{O}(n/(3/2)^i)$. And how large are the entries (or at least, the ones we care about)? They're at most $M \cdot (3/2)^i$, because the hop-lengths we care about go up to roughly $(3/2)^i$, and every weight is between $-M$ and M, so the total size of the weights of the paths we care about is M times the hop-length.

So if we do the $(\min, +)$ product, you could get bigger weights, because you are summing things that are both $M(3/2)^{i-1}$, and this could double it. But if at any point you get something bigger, you just set that to ∞ — you *cap* it. (This is a technical point.) Because the only things we care about are this size — you could get something bigger but those we don't care about.

So as we computee d_{i-1}, these weights are capped at $M \cdot (3/2)^{i-1}$ (in magnitude). Now that they're capped, we have our runtime for the $(\min, +)$-product from before. Here, M is $M(3/2)^i$, n is the same, and k and m are both basically $|S_i|$; so we get

$$M(3/2)^i n \, |S_i|^{\omega-1} .$$

This means the runtime is

$$M \left(\frac{3}{2}\right)^i \cdot n \cdot \left(\frac{n}{(3/2)^i}\right)^{\omega-1} .$$

So you get Mn^ω from teh parts not involving 3/2, and then you get $(3/2)^{i(\omega-2)}$ in the denominator — i.e.,

$$\frac{Mn^\omega}{((3/2)^i)^{\omega-2}}.$$

And this is great — we have $\omega \geq 2$, so this is at most $\widetilde{O}(Mn^\omega)$.

§4.7 The final output

So the runtime makes sense. Does what we computed make sense? Let's recap that. We've computed matrices d_i and $\overline{d_i}$ where we know, for each vertex and something in each sample, the distance between them, as long as the hop length is small enough.

Now after we've computed all these d_i's, we're going to build the matrix $\widetilde{D}$ that we want to compute.

Whenever we compute $d_i(v, s)$, we set $\widetilde{D}(v, s) = \min\{\widetilde{D}(v, s), d_i(v, s)\}$; similarly, whenever we compute $\overline{d_i}(s, v)$, we set $\widetilde{D}(s, v) = \min\{\widetilde{D}(s, v), \overline{d_i}(s, v)\}$. So $\widetilde{D}$ starts out as the adjacency matrix; and whenever we compute a new thing, we update with what we computed on this step.

And the only thing we need to store is $\widetilde{D}$.

Now, what do we need to show?

Claim 4.5 — With high probability, we have $d(u,v) = \min_{s \in V} \widetilde{D}(u,s) + \widetilde{D}(s,v)$.

How do we know this is the case? It depends on the hop length between u and v. So let's look at this hop-length.

Note that unlike Zwick's algorithm, we got to do this sampling all the way to the bottom, because it was cheap to compute. Zwick's algorithm had to stop at some point because the $(\min,+)$-product became very expensive. But here we're getting $\omega - 2$ instead of $\omega - 3$; which means we're not paying, and we get to go all the way to the bottom. So we have these distance matrices d_i for *every* choice of i.

Now if we look at u and v, we know $h(u,v) \in ((3/2)^i, (3/2)^{i+1}]$ for some i. But that means for that particular i, if I look at my uv path and I look at the middle third, for that particular i I know that S_i will hit the middle third with high probability. So for that i, S_i hits the middle third in some s. And we know $h(u,s)$ and $h(s,v)$ are then at most $(3/2)^i$, which means $d_i(u,s) = d(u,s)$ and $d_i(s,v) = d(s,v)$ (with high probability). And the left-hand sides equal $\widetilde{D}(u,s)$ and $\widetilde{D}(s,v)$ (we have $\widetilde{D} \leq d_i$, but $\widetilde{D} \geq d$; so because of the equality, we also get this).

And because of this, we have

$$d(u,v) = d(u,s) + d(s,v) = \widetilde{D}(u,s) + \widetilde{D}(s,v).$$

So this means that what we wanted is true — for every pair of vertices, we can find $d(u,v)$ by computing

$$\min_x \widetilde{D}(u,x) + \widetilde{D}(x,v).$$

(With high probability, but it can be derandomized, so that's okay.)

The query time is n — we don't know which S_i it is, and we don't know the hop length. So that means we basically have to try all the vertices in the graph. If I knew the hop length somehow, what would the runtime of the query be — can I save something? Suppose I knew, for my particular pair of vertices (u,v), that the hop length is somewhere in $[(3/2)^i, (3/2)^{i+1}]$. Then do I need to check through all the possible vertices for something to hit the path? No — I just need to check S_i. So if I knew something about the hop-length, then the only thing I need to check is S_i. And S_i is smaller than n — it's roughly $n/(3/2)^i$. So basically, if I knew the hop-length, my query time would be $n/h(u,v)$ instead. So the longer the path, the cheaper it would be to look it up. That's actualy very useful for some algorithms — if you can get a handle for how long the shortest path is (in terms of edges), you can query the distance oracle faster. The hardest things to query are the short paths (this is kind of bizarre, but it's true).

Student Question. *You said there was a corollary that would let us find negative cycles?*

Answer. For this, you need to figure out whether $d(u,u)$ becomes negative. And you can check that — the negative cycle is going to be simple, meaning it has at most n edges. So you can think of it as a path from u to u. If we're in the middle third, you hit it; and you check for each thing there whether $d(u,s) + d(s,u)$ becomes negative, which is basically the same thing.

Student Question. *Why do we need the cap?*

Answer. When you do the $(\min,+)$-product, you could have weights that double instead of getting multiplied by $(3/2)$; so you need to cap so that you don't have $2^i/((3/2)^i)^\omega$ instead.

This is basically the best-known distance oracle with *exact* distance queries. Later on, we'll see how to get much better ones for undirected graphs with approximation; you'll save much fewer bits and be able to query in constant time and get pretty accurate results. But they won't be exact.

§5 February 20, 2025

Today we'll talk about how to solve different path problems in faster than cubic time. We'll consider two path problems, and they'll basically be like shortest paths, except the measure of what makes a good path will be different. And then we'll see that even when the weights are arbitrary, these particular problems are easier than APSP, and you can use matrix multiplication to solve them even with very large weights.

The first problem set is due tonight (at midnight). The next pset will be out today and due in 2 weeks. Next week Virginia will be gone for a workshop, so Yael will give the two lectures; and then she'll be back.

§5.1 Bottleneck paths

Let's first define one of the problems; we'll start with bottleneck paths. For both of these problems, there'll be a general strategy of how you want to solve them; we'll see it with both examples.

Definition 5.1. A *bottleneck edge* on a path P is an edge $e \in P$ whose weight is the minimum among all the edges on that path P.

Why might you care about a bottleneck edge? For example, if you want to solve max-flow and you're pushing flow through a path, the weight of the minimum dge is what determines how much flow you can push along the path. And you've probably seen max-flow algorithms where you keep pushing flow along a path with maximum bottleneck.

A maximum bottleneck path betwen two vertices u and v is a path whose bottleneck edge has maximum weight (among all possible paths between u and v):

Definition 5.2. In a graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}$, a *maximum bottleneck path* from s to t is an s–t path with maximum bottleneck weight (among all paths from s to t).

Why might you care, besides max-flow? You can imagine your weights are heights of tunnels; and if you're going from s to t, you want to know what's the tallest truck you can route from s to t without crashing into a tunnel. There's many other applications. This is sometimes called the 'fattest path,' or many other things.

There are fast algorithms when s and t are given — you can solve it using Djikstra's algorithm, for example.

What we'll be interested in is the APBP problem, whre AP stands for 'all pairs' — we want to know is, for *all* pairs (s, t), what is the weight of the maximum bottleneck path?

Problem 5.3 (APBP)
For all $s, t \in V$, what is the weight of the maximum bottleneck path?

More generally, you might also want to compute these paths; we'll discuss how you can do that in a later lecture.

§5.2 From APBP to a matrix product

The first thing we want to think about with path problems is see if the path quality measure has some properties. For example, are the optimal paths simple — can they have loops? For this problem, we claim if you take a path that has some cycles (where it visits a vertex more than once), you can without loss of generality cut off that cycle and get a path that's at least as good.

Claim 5.4 — Optimal paths are without loss of generality simple.

Because the quality of the path is determined by the minimum weight of an edge on the path, if I kill off this cycle, that minimum weight can only increase, not decrease; so this only improves the path.

This is great, especially if we're working with matrices.

The second thing is, can we solve the problem if we only care about paths with only one hop, or only two hops? So for one hop, we want to define something like an adjacency matrix. For this problem, it's kind of like a max-min problem — you maximize over paths, of the minimum weights. (Shortest paths is a min-+ problem — you minimize over all paths, of the sum of weights.)

Question 5.5. How do we capture paths of a single edge?

This is defined by an 'adjacency matrix' for max-min (as opposed to min-+) — we define

$$A_{ij} = \begin{cases} w(i,j) & \text{if } (i,j) \in E \\ \infty & \text{if } i = j \\ -\infty & \text{if } (i,j) \notin E \end{cases}$$

So if ij is an edge, we just put its weight, as in a normal adjacency matrix. For the $i = j$ case, the point is that staying at a vertex should be free; and because we're maximizing, we want it to be ∞. And if it's not an edge, we want $-\infty$ — jumping to something you can't reach should really drop your bottleneck.

So this is our adjacency matrix, for at most one hop.

The next thing we're going to do is see what happens when we take *two* hops. For this, we have to define a matrix product. So for ≤ 2-hop paths, we want to define a matrix product. For shortest paths, this was easy — it was the $(\min, +)$-product. For transitive closure, it was the Boolean product. And so on. Here, it's the $(\max, \min)$-*product*:

Definition 5.6. The $(\max, \min)$-*product* of A and B, denoted $A \bullet B$, is defined as

$$(A \bullet B)_{ij} = \max_k \min\{A_{ik}, B_{kj}\}.$$

(The notation we use is a circled $\vee$.)

Claim 5.7 — For all i and j, we have that $(A \bullet A)_{ij}$ is the maximum bottleneck weight on an i–j path with at most 2 hops.

The point is here we have

$$\max_k \min\{A_{ik}, B_{kj}\}.$$

So if they're both edges, we're ust taking the minimum weight of those two; and then we're maximizing over all paths of this length. It could be that $i = k$, in which case I'm minimizing between ∞ and $w(k,j)$ (corresponding to a path of length 1); it's the same if $k = j$. And if A_{ik} corresponds to a non-edge, I'm taking the minimum of something with $-\infty$, which is just $-\infty$ (and when you take the maximum, you always prefer things that are bigger than $-\infty$).

The next step you've got to ask is, for this product, what do we need to compute? We know the optimal paths are simple, so in a certain sense, we need to compute the $(n-1)$st power of the adjacency matrix under this product — that would capture paths of length at most $n-1$, and because paths are without loss of generality simple, this will cover all the paths that we care about.

So we want to compute A^{n-1} under the $(\max, \min)$-product.

But now the question is:

Question 5.8. Can we compute A^{n-1} with successive squaring?

We can of course compute this by multiplying A by itself $n-1$ times, and that would take us $n-1$ iterations of the product. That would be pretty costly — the product itself takes at least n^2 time, so doing it $n-1$ times would take n^3 time. But if we could use successive squaring (as with transitive closure), then we would only pay a log.

In order to be able to apply successive squaring, we need this product to be associative — meaning we should have

$$(A \bullet B) \bullet C = A \bullet (B \bullet C).$$

So this is what we need. And this particular one actually happens to be associative — the quality of a path is just the minimum-weight edge, and whether you take the minimum by looking at the first edge vs. everything else, or up to some point and then everything else, it doesn't matter — the minimum is computed anyways. So this is actually associative.

And that means we can really do successive squaring.

So because this is associative, we immediately get:

Claim 5.9 — If the $(\max, \min)$-product is in $O(n^c)$ time, then APBP is in $O(n^c \log n)$ time.

Now, there's two things. The first thing is, in fact you can get rid of the $\log n$; there is a general theorem by Fischer and Meyer, who showed that if you can compute the matrix product over any semiring, then the transitive closure over that semiring can be computed in the same time (so the log disappears). We're not going to prove this, but we did prove it in class for transitive closure — we showed that if you can compute Boolean matrix multiplication in n^c time, then you can also do transitive closure in n^c time. That's matrix multiplication over the Boolean semiring, and this theorem generalizes that.

Theorem 5.10 (Fischer–Meyer)
If a matrix product $(\oplus, \otimes)$ over a semiring is in $O(n^c)$ time, then the transitive closure of A under this product is also in $O(n^c)$ time.

So in particular, the log gets killed. The way you prove it is similar in spirit to what we did for Boolean transitive closure, except it's a little more complicated because it's an arbitrary semiring.

And this $(\max, \min)$-product is the product over something called the *subtropical semiring* (which is basically max–min).

So the only thing we need to do to solve APBP is just to figure out a faster-than-cubic time algorithm for the product itself — what we need to do is figure out how to compute the $(\max, \min)$-product.

Goal 5.11. Compute $(A \bullet B)_{ij} = \max_k \min\{A_{ik}, B_{kj}\}$ in $O(n^c)$ time for some $c < 3$.

(The reason we want $c < 3$ is to show this is easier than shortest paths — for shortest paths, we don't know how to beat cubic, but for this one we do.)

§5.3 All-pairs earliest arrivals

We'll come back to this at the end; the next thing we'll do is look at our other problem and do the same steps.

This problem is as follows: Imagine you have lots of airports and flights (which consist of a departure and arrival city, and a departure and arrival time). You want to go from one airport to another by taking a

sequence of flights that form a valid itinerary — this means the arrival time of the ith flight should be less than the departure time of the $(i+1)$st. (You can also put some slack for layovers, but that can be encoded by increasing the arrival times a bit.) So a valid itinerary is a sequence of flights where the flight times are nondecreasing.

We can formalize this in the following way: the flights consist of a departure city and time, and an arrival city and time. A *valid itinerary* to get from s to t is a sequence of flights $f_1, f_2, \ldots, f_k$ such that dep-city$(f_1) = s$, arr-city$(f_k) = t$, and for all i, we have

$$\text{arr}(f_i) \leq \text{dep}(f_{i+1})$$

(and each flight arrives where the next one departs).

Now we can formulate this as a graph problem. Imagine you have a graph with a veertex for every airport (or city) on the left. And then you have your flights f on the right. And what you do is you put two edges — we draw a green edge $\text{dep}(f) \to f$, with deptime(f) as the weight. And then we draw an edge $f \to \text{arr}(f)$, with arrtime(f) as its weight. So now if we want to go from one city to another, you take some sequence of flights. And the paths should be *nondecreasing*.

And typically you want to minimize something — maybe how long it takes, or the number of flights. What we'll look at today is minimizing the arrival time — I want to get there as early as possible. So for every pair of cities, what's the (valid) itinerary that gets me to my destination as early as possible?

This has been studied since the 1950s (for flights and trains and so on).

What we're going to do now is forget about which vertices are flights and cities, and just consider a graph with some vertex set and weights corresponding to times. And we want to compute the best itineraries, which are the minimum nondecreasing paths — meaning the sequence of weights on the path form a nondecreasing sequence, and we're minimizing the weight on the last edge, which is the weight of the arrival time.

Definition 5.12. Given a graph $G = (V, E)$ and weights $w : E \to \mathbb{R}$, the weight of a nondecreasing path from s to t is the weight of the last edge. An *earliest arrival path* from s to t is a nondecreasing path (meaning the weights form a nondecreasing sequence) with minimum weight.

So we look at all paths from s to t with nondecreasing weights, and we take the one with the minimum last edge weight.

Problem 5.13 (APEA)

For every $s, t \in V$, what is the weight of the earliest arrival path from s to t?

If there is no path from s to t with a nondecreasing sequence of weights, then the earliest arrival is ∞ (you just never get there).

Student Question. *In the airplane case, is the graph sparse?*

Answer. For a sparse graph, if you only care about a pair of vertices, there's actually a linear algorithm — you can use Djikstra's with a different measure. But when you look at the all-pairs version, we don't know how to exploit sparsity, so we'll just imagine the graph is dense.

§5.4 APEP to a matrix product

Now let's do the same steps as before.

First, we want the optimal paths to be without loss of generality simple. Is that the case? If I take a nondecreasing path from s to t with minimum weight and it has some cycle, I can kill off the cycle and still have a nondecreasing path with the same weight; so this is good.

Now let's do the adjacency matrix for earliest arrivals. If there's an edge from i to j, we should put the weight of that edge, as usual. If $i = j$, what should we do? We're actually not even going to consider this — we'll see why later. And if there's no edge, we'll set it to $+\infty$ — we'll be minimizing, so we want in the end that paths that don't exist should be the worst possible, which is ∞. So we define

$$A_{ij} = \begin{cases} w(i,j) & \text{if } (i,j) \in E \\ \infty & \text{otherwise.} \end{cases}$$

Now let's consider a product. The product we'll use is:

Definition 5.14. The $(\min, \leq)$-*product* of A and B, denoted $A \bullet B$, is defined as

$$(A \bullet B)_{ij} = \min\{B_{kj} \mid A_{ik} \leq B_{kj}\}.$$

(If there's no such k, we set this to ∞.)

(The symbol we use in class is a circled $<$ sign.)

Now what happens if I take this adjacency matrix and look at $(A \bullet A)_{ij}$? This is

$$(A \bullet A)_{ij} = \min\{A_{kj} \mid A_{ik} \leq A_{kj}\}.$$

So if I take a path from i to k to j, then it'll be the minimum last edge of a path of length 2 such that the weight of ik is at most that of kj. (If one of these is not an edge, we get ∞.) So this is the minimum weight of a nondecreasing path of *exactly* two hops.

Now, given that product, we want to compute A^{n-1} (but actually here we also need A^1, A^2, ...) under the $(\min, \leq)$-product.

Question 5.15. Can we compute A^{n-1} under the $(\min, \leq)$-product using successive squaring — i.e., is this product associative?

The answer is actually no — it's *not* associative. As an example, let's take a graph with $1 \to 2$ of weight 8, $2 \to 3$ of weight 9, $3 \to 4$ of weight 10, $2 \to 5$ of weight 1, and $5 \to 4$ of weight 1. And let's suppose we want to go from 1 to 4; so we consider

$$(A \bullet (A \bullet A))_{14} \text{ and } ((A \bullet A) \bullet A)_{14}.$$

The latter is 10 — first I minimize over paths of length 2, so I get ∞ from 1 to 5 and 9 from 1 to 3. And then when I apply the secon dpart, I'll get $\infty \not\leq 1$, so $1 \to 5 \to 4$ is not valid; meanwhile $9 < 10$, so I get 10. But if I do it the other way, th ebest nondecreasing path to go from 2 to 4 is actually 1 (while going from 2 to 4 would give 10). And then when I try to do the second part, taking the $(\min, \leq)$-product of 8 and 1, we have $8 > 1$, so we actually get ∞. So there's actually a big difference — the order in which you do the product makes a big difference.

So now we have to properly define what we mean by the power of the matrix.

Definition 5.16. We define $A^{i+1} = A^i \bullet A$.

So inductively we tack things on to the right — because you want to get here first and then tack on an extra edge, and so on.

§5.5 Short paths-long paths

Because it's not associative, we actually want to take the minimum over all the powers — A^1, A^2, …, A^{n-1}.

That's too expensive. But we have more tricks from the previous lectures.

We can't afford to do n successive products, and we can't do successive squaring. But there's something we did in the previous lectures that we can definitely use here.

We can use hitting set arguments! Last time, we looked at short-path long-path techniques. We said if the paths are very long, we can find this hitting set that hits them somewhere. And once we have that, all we need is a SSSP algorithm; then we get to minimize over the hitting set, and that's cheaper. And then for the short paths, we just do this — we just calculate this product a small number of times. And then we take the minimum of the two approaches.

So we'll let s be some parameter. And we'll say *short paths* have at most s hops, and *long paths* have more than s.

For short paths, we compute A, A^2, A^3, …, A^s. The way we do this is if we had a $(\min, \leq)$ algorithm that runs in n^c time, this will only cost us sn^c. So let's suppose the $(\min, \leq)$-product has an $O(n^c)$ time algorithm, where hopefully $c < 3$. Then the short path case will take us $O(sn^c)$ time (if we do it in the silly way, where we compute all these and then take the componentwise minima; and then we'll get the best paths among those with at most s hops).

The second part is the long paths. For this, we take $S \subseteq V$ to be a random sample of size $\Theta(\frac{n}{s} \log n)$. And then for every pair of vertices, we imagine that we pick some minimum nondecreaing path with at least s hops; and then this hitting set S will with high probability hit every one of these at most n^2 paths. So with high probability, S hits some long earliest arrival path for all $u, v \in V$ with such a path.

So we have our picture where S is a blob, and for every u and v with a long path, there's some nondecreasing path and S contains one of its nodes. And now we need to have some sort of algorithm that'll compute, for every pair of vertices, the minimum nondecreasing path that goes through this set.

So for a fixed $s \in S$, we want a single-source algorithm from s. What does this need to do?

> **Goal 5.17.** Given s, for every u and v, we want to compute the earliest arrival path weight from u to v where the path goes through s.

Once we have such an algorithm, if it's super fast, then we can run it on every vertex in S; and for all u and v, you take the minimum over all $s \in S$ of the earliest arrival path that goes through s.

So it is *not* from the source to v and u to the source; but it is a path that *goes* through the source, somehow. (This isn't just two single-source computations.)

> **Claim 5.18 —** This problem can be solved in $\widetilde{O}(n^2)$ time (for any fixed s).

We are not going to prove this, because it will be on the second problem set.

Once we have this for a single s, then for all u and v, we take

$$\min_{s \in S} \mathrm{ea}_s(u, v),$$

where this denotes the earliest arrival weight from u to v through s. And this computation will just take us $n^2 |S| = \widetilde{O}(n^3/s)$ time (the size of our sample is $n/s \log n$, and the log gets put into the $\widetilde{O}$).

So for short paths we have runtime sn^c, and for long paths we have runtime $n^3/s + |S| n^2 = n^3/s$ (where the second term is to do paths through each $s \in S$, from pset 2).

So we should set these equal; this means we want $s = n^{(3-c)/2}$, and we get a runtime of $\widetilde{O}(n^{(3+c)/2})$. So we get the average of the product exponent and 3.

So our task is now:

Goal 5.19. Compute $(A \bullet B)_{ij} = \min\{B_{kj} \mid A_{ik} \leq B_{kj}\}$ in time $O(n^c)$ for some $c < 3$.

And given that we can do this in n^c time, we immediately get $n^{(3+c)/2}$ time for APEP; and if $c < 3$, then $(3+c)/2 < 3$ as well. So as long as we get a subcubic algorithm for the product, we also get one for the all-pairs problem. It's not as good — we lose something, because $(3+c)/2 > c$ — but it's still subcubic.

And then, as we'll see, we *can* actually achieve $c < 3$. Later on people showed that you don't actually need this $(3+c)/2$ loss if you're more clever — if you also consider sparse matrices and rectangular matrices and various things, you can actually get the same runtime of n^c for the product and the APEP problem.

§5.6 More products

Now we'll show how to actually compute these products in subcubic time. The moment we get those, we'll get the all-pairs paths problems for free, by these reductions.

The way we're going to do that is by defining two more products and going between them.

Definition 5.20. We define $(A \bullet B)_{ij} = \max\{A_{ik} \mid A_{ik} \leq B_{kj}\}$ and $(A \bullet B)_{ij} = \#\{k \mid A_{ik} \leq B_{kj}\}$.

(These are called the $(\max, \leq)$ and dominance product, respectively. In class we use a circled $>$ and a circled d.)

It turns out that $(\min, \leq)$ and $(\max, \leq)$ are equivalent — if you can solve one, you can solve the other in the same runtime. And if $\bullet$ is in n^c time, then both $(\max, \leq)$ and $(\min, \leq)$ are in $n^{(3+c)/2}$ time; and then $\bullet$ is also in $n^{(3+c)/2}$.

So in other words, you can reduce $\bullet$ to $(\min, \leq)$ or $(\max, \leq)$ (which are the same), and those to the dominance product. So if we can solve the dominance product in subcubic time, we can actually do all of them.

The dominance product was the one shown first; it's by Matousek from 1991, and solves the problem in $c = (3+\omega)/2$ time. We'll see these first. Then we'll show that $(\max, \leq)$ and $(\min, \leq)$ are the same; then we'll show the reduction from dominance to those; and those to $\bullet$ is actually pretty immediate.

§5.7 Min–$\leq$ and max–$\leq$

Let's start with $(\min, \leq)$ and $(\max, \leq)$ — why are these the same? It turns out that

$$-(-B^\intercal \bullet -A^\intercal)(j,i) = (A \bullet B)_{ij}.$$

SO if I want to get the $(\max, \leq)$ product of A and B, I can compute it like on the left-hand side. This is because hte left-hand side is

$$-\min\{-A^\intercal_{ki} \mid -B^\intercal_{jk} \leq -A^\intercal_{ki}\}$$

by definition. And now this min becomes a max if I move the $-$ inside; so this becomes

$$\max\{A^\intercal_{ki} \mid \cdots\}.$$

Now $A^\intercal_{ki} = A_{ik}$, and we can also flip the inequality and remove the minus sign; so we get

$$\max\{A_{ik} \mid A_{ik} \leq B_{kj}\},$$

which happens to be exactly $(A \bullet B)_{ij}$.

So $(\min, \leq)$ and $(\max, \leq)$ are actually the same (up to some negations and transpositions). So this is fine.

§5.8 Computing $(\max, \min)$ from $(\max, \leq)$

Next we'lls ee how to compute the $(\max, \min)$ product using $(\max, \leq)$.

We can write

$$\min\{A_{ik}, B_{kj}\} = \begin{cases} A_{ik} & A_{ik} \leq B_{kj} \\ B_{kj} & B_{kj} \leq A_{ik}. \end{cases}$$

This immediately means that

$$(A \bullet B)_{ij} = \max\{(A \bullet B)_{ij}, (B^\intercal \bullet A^\intercal)_{ji}\}.$$

So the $(\max, \min)$ product can be computed with two $(\max, \leq)$ products. That means in order to solve $(\max, \min)$-product, it suffices to solve this one. (We don't know if the opposite reduction exists, but we do have one in this direction.)

§5.9 Dominance product

Now it remains to solve dominance product, and show how this leads to the $(\max, \leq)$ product. We'll first see how to solve dominance product; once we have that, it'll be easier to see the reduction from that to $(\max, \leq)$.

Theorem 5.21 (Matoušek 1991)
The dominance product can be computed in $\widetilde{O}(n^{(3+\omega)/2})$ timee.

Let's imagine we have two matrices A and B (which are $n \times n$; we'll assume wihtout loss of generality that they're integers). And we want to compute the number of k's for which $A_{ik} \leq B_{kj}$ (we say that B_{kj} dominates A_{ik}).

What's common between A_{ik} and B_{kj} in this inequality is k. So we're going to take, for each *fixed* k, all the entries of A in the kth column, and all the entries of B in the kth row. And we'll put that together into a set, and we'll sort this set. So for each fixed k, we define L_k as a sorted list of $\{A_{ik}\}_i \cup \{B_{kj}\}_j$. For example, if

$$A = \begin{bmatrix} 2 & 5 \\ 1 & 7 \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} -1 & 7 \\ 8 & 100 \end{bmatrix}$$

and $k = 2$, then we take the column $(5, 7)$ from A and the row $(8, 100)$ from B; so when we sort them, we get

$$L_2 = [5, 7, 8, 100] = [A_{12}, A_{22}, B_{21}, B_{22}].$$

In general, these can be intermixed — for any fixed k, you put them all together and sort them, and you get a sorted list. The reason we want to sort them is that we care about the comparison for fixed k.

So this is L_k — it's some sorted list of A's and B's with fixed k. And what we'll do now is split L_k into pieces. The length of L_k is $2n$ (so it took us $n^2 \log n$ time to sort the whole thing for each k). And now we'll split it up into pieces, following things we've done in the past — we'll split it into pieces of size t, where t is some parameter. So we've split L_k into $2n/t$ buckets of size t. (This is in the sorted order — so every element in the first bucket is less than or equal to every element in the second bucket, and so on.)

Let's look at th bth bucket, which w'ell call L_{kb} — all th et stuff in the bth bucket. And I know that for every b, everything in L_{kb} is less than or equal to everything in $L_{k(b+1)}$, and so on (because we have this sorted order).

Now I'm going to consider these comparisons $A_{ik} \leq B_{kj}$ in two cases. One case is when A_{ik} and B_{kj} are in the same bucket (for k), and the other case is when they're in different buckets.

So there's two cases for $A_{ik} \leq B_{kj}$.

Case 1 (A_{ik} and B_{kj} are in different buckets of L_k). They're both in the same list L_k, but in this case they fall in different buckets. This means A_{ik} is in some L_{kb} and B_{kj} is in some $L_{kb'}$, and we know that $b' > b$ — because $A_{ik} \leq B_{kj}$ and they're in different buckets, so B_{kj} has to be in a later bucket (because the buckets are sorted).

Case 2 (A_{ik} and B_{kj} are in the same bucket L_{kb} (for some b)). Now I claim we can compare A_{ik} and B_{kj} in the *same* bucket much more cheaply. How do I do that?

Let's say our output is some D_{ij} (so D is our desired output matrix). We originally set it to the 0 matrix; and what we're goig to do now is say, there's n^2 choices for i and k. For each, A_{ik} is in some bucket L_{kb} (and we know which b it is). Then for *every* B_{kj} in the same bucket, if $A_{ik} \leq B_{kj}$, we increment $D_{ij} \leftarrow D_{ij} + 1$. So for every entry A_{ik} I figure out what bucket of L_k it's in; for every other element of the bucket, I check if it's from B, and if it is then I increment the count. This is cheap because the size of the bucket is very small — we said it's t, where t is small. So the time this takes is $O(n^2 t)$ (to handle everything that's in the same bucket); if t is much less than n, this is subcubic.

So now we have to deal with the case where they're in different buckets. And this is where we'll use matrix multiplication. For every single bucket $b \in [2n/t]$ (there's $2n/t$ choices of buckets), we're going to define a matrix A^b by

$$A^b_{ij} = \begin{cases} 1 & \text{if } A_{ik} \in L_{kb} \\ 0 & \text{otherwise,} \end{cases}$$

and we define

$$B^b_{kj} = \begin{cases} 1 & \text{if there exists } b' > b \text{ such that } B_{kj} \in L_{kb'} \\ 0 & \text{otherwise.} \end{cases}$$

In other words, the bucket for which B_{kj} appears is after the bucket b.

What is $(A^b \cdot B^b)_{ij}$ (taking the integer product)? This is the number of k such that $A_{ik} \in L_{kb}$ and $A_{ik} \leq B_{kj}$ and B_{kj} is not in L_{kb}. So the comparison is correct, but it's not in the same bucket (because it's in a later bucket).

So for every bucket, we can compute this product; and then we get the number of k where they're in different buckets and the first is in bucket b. And then we take

$$\sum_b A^b \cdot B^b,$$

and this is going to give us what we want — this is the sum over the case where A_{ik} and B_{kj} are in different buckets.

What's the runtime? The number of terms here is n/t, and each takes n^ω (and the sum is just $n/t \cdot n^2$, so it gets dominated by this). SO this is $O(n^{1+\omega}/t)$.

And the total is $n^2 t + n^{1+\omega}/t$; when we have such a tradeoff, we know the final exponent is the average of 3 and ω. So the final runtime is $n^{(3+\omega)/2}$.

What we've done here is we've gotten the dominance product in $n^{(3+\omega)/2}$.

§5.10 Dominance to min-leq

We have not shown how to get $(\min, \leq)$ from dominance, but we'll give a 2-minute summary. Once you knwo the sorting trick, this should not be too bad.

The dominance product in particular tells us, is there some k with $A_{ik} \leq B_{kj}$? (Since it in particular counts the number of k for which this is the case.) But in $(\min, \leq)$, we want to find the *minimum* B_{kj} such that

this comparison is true. We know how to deal with minima, because we did it above — we use sorting. So here what we're going to do is we just sort the matrix B — for every column j of the matrix, we're going to sort it. So we take B, and we take the jth column, and you sort it. Then you get a list L_j which is just the sorted order of column j. We split it into buckets of size t.

Then we define two matrices, and we compute their dominance product. We have the matrix A, and a matrix B^b (for bucket b) defined as

$$B^b_{kj} = \begin{cases} B_{kj} & \text{if } B_{kj} \text{ is in the } b\text{th bucket of } L_j \\ -\infty & \text{otherwise.} \end{cases}$$

And then we do the dominance product $(A \bullet B^b)_{ij}$, and this will tell us the number of k such that $A_{ik} \leq B_{kj}$ and B_{kj} is in bucket b.

Now in particular, we'll find the smallest bucket for which this comparison is true. And this list is sorted; so this smallest bucket tells us where there is some B_{kj} which is at least A_{ik}.

So for every (i, j), we find the smallest b such that $(A \circ B^b)_{ij}$ is nonzero. This will tell su the location of B_{kj} for our minimization problem — we want to minimize B_{kj}, and we know where it is in the sorted order. Once we know that, we just brute-force — you try all the possible things in this bucket and find the smallest one for which this inequality holds.

Computing all these dominance products will cost us $n/t \cdot n^c$ where c is the dominance product time. And the brute force is over all n^2 choices of (i, j), where we brute force over a bucket of size t; so that's $n^2 t$. And again, we get $n^{(3+c)/2}$.

§6 February 25, 2025 — Paths, successors, and witnesses

Last week we talked about different shortest paths you might want to compute. We saw earliest arrival paths, where we computed the $(\min, \leq)$-product; we saw bottleneck paths; and before that, we saw regular APSP, with the regular graph distance. We saw how to compute all of these through different matrix products.

In your algorithms class, when we learn about BFS or Djikstra and want to find shortest paths, we don't just get the distances, but also the paths themselves (you get the predecessors, and can reconstruct the paths from that). We'd like to be able to do the same thing here. So we'll talk about that today.

§6.1 A general matrix product

We'll start by defining something that generalizes all the things we saw before, so that we can talk about them all at once.

Definition 6.1. Given a binary operation $\odot : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ and a graph G with edge weights $e : E \to \mathbb{Z}$, we define

$$w_\odot(i_1 \to i_2 \to \cdots \to i_t) = ((w(i_1, i_2) \odot w(i_2, i_3)) \odot w(i_3, i_4)) \odot \cdots.$$

We need the parentheses because $\odot$ might not be associative (e.g., the one for earliest arrival shortest paths wasn't).

Definition 6.2. We define $d_\odot(i, j) = \min_P w_\odot(P)$ (where P is a path from i to j).

We'll assume that shortest paths are simple, so that this is well-defined.

Now we'll turn this into a matrix product, as before:

Definition 6.3. We define the $(\min, \odot)$-product of two matrices A and B as

$$(A \odot B)[i,j] = \min_k (A[i,k] \odot B[k,j]).$$

All these operations can be represented in this way — in bottleneck paths we have a $(\max, \min)$-product, but we can turn this into a $(\min, \odot)$-product (usually when we have a max,these operations are of a form where we can add minuses to turn it into a min).

What about Boolean matrix multiplication? We can also represent it in this way — there we're just seeing if there's a place where they're both 1, so we can think of it as taking a maximum. So we can think of Boolean matrix multiplication as

$$(A \cdot B)[i,j] = \max_k \{A[i,k] \cdot B[k,j]\}.$$

And to turn this into a $(\min, \odot)$-product, we just negate one of the matrices.

§6.2 Predecessor and witness matrices

Now we're going to try to define a *predecessor matrix* for this operation. When we compute distances we just need n^2 information; but paths could have length n, so storing the entire paths could take n^3. That's too much, so instead we'll want to store *predecessors*.

Definition 6.4. A matrix P is a *predecessor* for the all-pairs $(\min, \odot)$-product if for every pair (i,j),

$$d_\odot(i,j) = d_\odot(i, P[i,j]) \odot w_\odot(P[i,j], j).$$

So what this is saying is that on the shortest path from i to j, $P[i,j]$ should return the last vertex on this path (before j). Then $d_\odot(i, P[i,j])$ gives us the weight of the shortest path up to this last vertex, and the last term gives the weight of the last edge.

Given the predecessor matrix, how do we reconstruct shortest paths?

Proposition 6.5

Given P, we can recreate the shortest path from i to j in time linear in the number of hops on the path.

Here $P[i,j]$ gives us the last vertex $w_1 = P[i,j]$ on the path; then to get the one before it we take $w_2 = P[i, w_1]$; and then we take $w_3 = P[i, w_3]$; and so on. So we can reconstruct the path going backwards.

This gives us the second-*last* vertex. Sometimes we want to talk about the *second* vertex, which we call the *successor* (the one immediately after i). We can define this in the same way:

Definition 6.6. A *successor* is a matrix S such that for all (i,j), we have

$$d_\odot[i,j] = w(i, S[i,j]) \odot d_\odot(S[i,j], j).$$

When can we have an issue with a successor vs. predecessor matrix? If the matrix product isn't associative — in earliest arrivals, the successor matrix won't be helpful (if you optimize from the left end, things go wrong). So we only care about the successor matrix when things are associative (as in all the examples we've see except earliest arrival).

A similar definition, which we'll use to get these matrices, is the *witness matrix*, which essentially tells us, in a matrix product $(A \odot B)$, where the minimum was achieved.

Definition 6.7. A *witness* W for a matrix product $(A \odot B)$ is a matrix storing, for each (i, j), an index k such that $(A \odot B)[i, j] = A[i, k] \odot B[k, j]$.

By the end of today, we'll hopefully prove that computing this witness matrix takes roughly the same time as computing the actual matrix product, up to a few log factors.

Theorem 6.8

If there is an algorithm for the $(\min, \odot)$ product running in $T(n)$ time, then there is a randomized algorithm that computes the witness matrix W (with high probability) in time $O(T(n) \log^3 n)$.

§6.3 Applications of the witness matrix

This is what we'll prove in the end. But first let's see why witnesses are even helpful — how these witness matrices can help us get predecessors or successors and compute the paths we care about.

§6.3.1 Seidel's algorithm for APSP

A few weeks ago, we saw Seidel's algorithm for computing APSP in unweighted, undirected graphs. How did Seidel use BMM to compute APSP? The idea is that by squaring, we halve the lengths of the paths, but we lose ± 1 information; and Seidel used BMM to fix the parity.

Here we'll show that if we can get witnesses in $T(n)$ time, then we can also get this predecessor matrix in the same time. The idea is that if k is a neighbor of j, then $d[i, j]$ and $d[i, k]$ differ by at most 1. And we have

$$d[i, k] = d[i, j] - 1$$

if and only if k is the predecessor of j on the shortest path from i to j (we'll write $i \rightsquigarrow j$ to denote this path).

And in the homework, we saw that we could say something weaker than requiring an exact equality. We know $d[i, j]$ and $d[i, k]$ differ by at most 1; so it's enough to require that these are equal mod 3. In other words, $d[i, k]$ can only take values that are $d[i, j]$, $d[i, j] + 1$, or $d[i, j] - 1$. So it has three different options; and that means we can look at this mod 3, instead of just looking for equality. So if

$$d[i, j] \equiv d[i, j] - 1 \pmod 3,$$

then they're actually equal (not just mod 3), and that means k is a predecessor of j on this path.

We're going to use this fact to compute three different matrix products. For each $s \in \{0, 1, 2\}$ (representing the three values you could have mod 3), we'll define a matrix

$$D^{(s)}[i, k] = \begin{cases} 1 & \text{if } d[i, k] \equiv s - 1 \pmod 3 \\ 0 & \text{otherwise.} \end{cases}$$

So D is the distance matrix we've computed from Seidel's algorithm, but only for the value mod 3 that we care about — or really, it's an indicator for what things are mod 3.

Now we compute the product $D^{(s)} \cdot A$ (where A is the adjacency matrix).

To be more explicit, we start by computing all the distances $d[i, j]$ using Seidel's algorithm. And as we had before, A is the adjacency matrix. Now we're going to compute this Boolean matrix product, and we'll let $W^{(s)}$ be the witness matrix for this product $D^{(s)} \cdot A$.

So now we want to use W to find predecessors for our path. What is $W^{(s)}[i,j]$? For a specific pair (i,j), we only care about one value of s, namely $s = d(i,j) \pmod 3$. And for this s, this witness matrix gives us a point k such that this product $D^{(s)} \cdot A$ gave us 1, meaning

$$D^{(s)}[i,k] \cdot A[k,j] = 1.$$

That means both $A[k,j]$ and $D^{(s)}[i,k]$ are 1. The information $A[k,j] = 1$ tells us that k is a neighbor of j. And $D^{(s)}[i,k] = 1$ tells us that

$$d[i,k] \equiv s - 1 \pmod 3 \equiv d[i,j] - 1 \pmod 3.$$

And from what we had before, these things together tell us that k is the predecessor of j on $i \rightsquigarrow j$.

So now using this witness matrix, how can we construct a predecessor matrix for this problem? We want to define $P[i,j]$ for every i and j. And we can do this by taking the correct s and taking the entry of that witness matrix at $[i,j]$ — so

$$P[i,j] = W^{(d[i,j] \bmod 3)}[i,j]$$

is the predecessor matrix for (regular) shortest paths.

§6.3.2 Successors for transitive closure

Now we're going to try to use the witness matrix to get predecessors for transitive closure.

To get transitive closure, we did something a bit simpler than Seidel's algorithm — it's Seidel's algorithm without trying to fix the parity issues. We just successively square the adjacency matrix $\log n$ times.

In transitive closure, when we're looking for predecessors, what are we looking for? Here we're actually going to look for successors; what we want is that the (i,j)th entry of the successor matrix to be the next vertex k on *some* (simple) path between i and j (it doesn't have to be a shortest path). So we want to know that if i can reach j, after one step k can reach j. And if there's no path, then $S[i,j]$ can be something arbitrary.

Now we're going to see an algorithm to try to compute these successors.

We'll start with our generalized adjacency matrix $A^{(0)}$; this is the adjacency matrix, where I also add 1's on the diagonal (because vertices can reach themselves) — this represents which vertices can reach each other in at most one hop (we write $i \rightsquigarrow j$ in ≤ 1 hop).

We'll start with $S^0[i,j] \leftarrow j$ if $A^{(0)}[i,j] = 1$. So this is our successor matrix for vertices which can reach each other in one step; the next step of the path here is just going to be j.

Now we're going to repeat this $\log n$ times. So for $k = 1, 2, \ldots, \log n$, we'll start our squaring — we'll have

$$A^{(k)} = A^{(k-1)} \cdot A^{(k-1)},$$

and we'll call $W^{(k)}$ the witness from this operation. For all i and j, if $A^{(k-1)}[i,j]$ was 1 (so in the previous iteration we already knew i could reach j, meaning we already had a witness), then we leave S to be the same — we define

$$S^{(k)}[i,j] = S^{(k-1)}[i,j].$$

Otherwise, we're going to set

$$S^{(k)}[i,j] = S^{(k-1)}[i, W^{(k)}[i,j]].$$

The point is now we've found a new path, so we want to set the successor to be the next step from i to the witness.

And in the end, we set $T \leftarrow A^{(\log n)}$ and $S \leftarrow S^{(\log n)}$.

We'll now prove that this works, inductively.

Claim 6.9 — We have that $S^{(k)}[i,j]$ is a successor for $i \rightsquigarrow j$ if $d[i,j] \leq 2^k$.

Proof. When we start with 0, that's true — that's how we initialized $S^{(0)}$. Now if it's true for pairs of points that are up to distance 2^{k-1} — if $d[i,j] \leq 2^k$, then $W^{(k)}[i,j]$ is going to be some point w such that there's some path $i \rightsquigarrow w \rightsquigarrow j$ where each of the two segments is at most 2^{k-1} (because each was in $A^{(k-1)}$ from before). So then $S^{(k-1)}[i,w]$ is going to give us a successor for this path from i to w; and that's a successor for the path from i to j. And that's exactly what we're putting into $S^{(k)}$. □

§6.3.3 Zwick's algorithm for APSP

We'll see one last way to use witness matrices before we start discussing how to compute them — now we're going to use Zwick's algorithm for (weighted) all-pairs shortest paths.

Zwick's algorithm is about hitting the middle third — we think about paths of length up to $(3/2)^{i+1}$, and we're going to look for some point in the middle where its distance from both endpoints is less than $(3/2)^i$.

So let's assume we can get the witness matrix for the $(\min,+)$-product (with entries in $\{-M,\ldots,M\}\cup\{\infty\}$) in $\widetilde{O}(Mn^\omega)$ time (this is the amount of time it took us to get the $(\min,+)$-products; and we're assuming we can do the same with witnesses).

We'll now see how to use these witnesses to get the actual paths. It's basically the same idea as with transitive closure — we'll initialize a successor matrix for points 0 or 1 hop apart. Then we'll use Zwick's algorithm where we compute distances to our hitting sets, and each time we update our successors using the previous successor from i to the witness.

So at each point in Zwick's algorithm, when we update our distance $d[i,j]$ as $d[i,w]+d[w,j]$, we'll also update the successor matrix as $S[i,j] = S[i,W[i,j]]$ (where $W[i,j]$ is the witness w we got when computing this specific $(\min,+)$, and $S[i,W[i,j]]$ is the successor from i to there).

§6.4 Computing witnesses

Now for the time we have left, we'll try to prove the theorem stated earlier, which says the witnesses are actually not harder to compute than just the matrix product.

§6.4.1 Unique witness matrices

We'll start with a simpler case, where there's only a unique witness — if there's only *one* spot where this value is achieved, then we want to return that; and if there's more than one, we don't care (it can return an arbitrary value).

Definition 6.10. A matrix U is a *unque witness matrix* for the $(\min,\odot)$-product $A \odot B$ if $U[i,j]$ is a value k such that

$$(A \odot B)[i,j] = A[i,k] \odot B[k,j]$$

whenever k is a *unique* minimizer.

If there's more than one k for some (i,j), then $U[i,j]$ can be arbitrary; we don't know what it's going to be.

Let's start by computing the unique witness for Boolean matrix multiplication. This we can actually do by integer multiplication.

Say we're trying to multiply two matrices; and we only care about getting the witness where we only have *one* place where the 1's line up (everything else cancels out). Then we can take A and instead of 1's, we put in the column number — so we define

$$A'[i,k] = kA[i,k].$$

Now the unique witness matrix U is just $A'B$, where instead of Boolean matrix multiplication now we're doing *integer* matrix multiplication. So here we can just do this cool trick.

But actually, in *any* matrix product we can pretty easily compute the unique witness.

Lemma 6.11

If we can compute the $(\min, \odot)$-product in $T(n)$ time, then we can compute the unique witness matrix in $O(T(n) \log n)$ time.

So with BMM we did this trick where the row gave us an indication of what the unique witness was; but here we don't know how exactly this operation is working. So what's something we can do to try to find the unique witness?

We care about (i,j) where there's only one k that minimizes it, and we want to find it. So we'd like to do some sort of binary search, where we look at different subsets of the columns, and see when k is in them and not; and we'll try to find k using that.

Notation 6.12. For a subset $S \subseteq [n]$, we define $A[\bullet, S]$ to denote A restricted to the columns in S, and $B[S, \bullet]$ to be the same with rows.

So if $S = \{1,2,3\}$, we're taking the first three columns of A and first three rows of B.

How should we start searching for our unique witness k? Let's look at the binary representation of k, and try to figure it out bit by bit. So for $b = 1, \ldots, \log n$, we'll take

$$S_b = \{1 \le k \le n \mid b\text{th bit of } k \text{ is } 1\}.$$

Now if we look at $A[\bullet, S_b] \odot B[S_b, \bullet]$, what information do we get? We'll write

$$C_b = A[\bullet, S_b] \odot B[S_b, \bullet]$$

to denote this product. And then we'll set

$$W[i,j]_b = \begin{cases} 1 & \text{if } C_b[i,j] = C[i,j] \\ 0 & \text{otherwise} \end{cases}$$

(where $W[i,j]_b$ is the bth bit). The point is that if $C_b[i,j] = C[i,j]$, then the unique witness k was in S_b (otherwise we'd get something worse); so that tells us the bth bit of that witness is 1. So if we do thsi for all b, then we get $W[i,j]$ for all (i,j) with a unique witness.

So computing the *unique* witness added one log factor; now we'll see that getting from non-unique to unique will add another, and getting 'with high probability' will require us to add one more.

§6.4.2 Non-unique to unique witnesses

What can we do if we have too many witnesses? The idea is we're going to try to sample and get to a point where we have exactly one witness.

We'll do this in two steps. First we'll assume we *know* how many witnesses we have.

Claim 6.13 — Assume a pair (i,j) has c witnesses, where $\frac{n}{2^{s+1}} \leq c \leq \frac{n}{2^s}$. Let S consist of 2^s elements picked independently at random from $[n]$ (with replacement). Then S contais a *single* witness for (i,j) with probability at least $1/2e$.

Proof. Let W be the set of witnesses for (i,j). What's the probability that $|S \cap W| = 1$? One way to compute this is we've picked 2^s different elements; so we can ask, which element happens to be the one that falls in W? We have 2^s options to pick which one; and then there's a $\frac{c}{n}$ probability this specific element is a witness. And what's the probability that everything else *isn't* a replacement? Each thing is *not* a witness with probability $1 - \frac{c}{n}$, and we have $2^s - 1$ such elements. So

$$\mathbb{P}[|S \cap W| = 1] \geq 2^s \cdot \frac{c}{n} \cdot \left(1 - \frac{c}{n}\right)^{2^s - 1} \geq 2^s \cdot \frac{1}{2^{s+1}} \cdot \left(1 - \frac{1}{2^s}\right)^{2^s - 1}.$$

The first two terms are at least $1/2$, and the last is at least $1/e$. □

So if we sample 2^s elements, then with constant probability we'll have hit exactly one witness from the set that we care about.

§6.4.3 Proof of theorem

Let's suppose we want to compute a witness matrix for (A,B). We only have to try $\log n$ possible values of s; so we'll try all of them, and sample, so that with constant probability we get a unique witness; and then we'll run the unique matrix computation. And then we'll have to do this $\log n$ times to get high probability.

So we'll initialize $W \leftarrow \infty$ (to denote that we haven't filled it out yet), and compute $C \leftarrow A \odot B$. Then for each $s = 0, \ldots, \log n$, we'll sample S to consist of 2^s elements of $\{1, \ldots, n\}$ with replacement. Now how do we use this to try to get to a unique witness case? We'll define W' to be the unique witness of $A[\bullet, S] \odot B[S, \bullet]$ — so we compute this operation and get the unique witness matrix. So every (i,j) that had a unique witness in this will appear here in W'.

But because W' can also contain arbitrary values, we have to check that we actually got the value we care about. So for all i and j, we check whether the unique witness in W' is actually a good witness — we check that

$$A[i, W'[i,j]] \odot B[W'[i,j], j] = C[i,j].$$

If this is true, that tells us $W'[i,j]$ is exactly the witness we're looking for. So if this is true, then we've found a witness, and we can put it now in W — so we put $W[i,j] \leftarrow W'[i,j]$.

And then we're going to repeat this $\log n$ times (for each value of s).

Why does this work? First, can $W[i,j]$ ever have a *wrong* value? No — if $W[i,j]$ ever gets a value, then that value *will* be a witness, because we ran the above check. So we just need to make sure that $W[i,j]$ does get filled (for every (i,j)).

For each (i,j), we know there's some s for which the number of witnesses is between $n/2^{s+1}$ and $n/2^s$. So from the claim, with probability $1/2e$, our set S will contain a unique witness; and then from the unique witness-finding lemma, we'll find that unique witness. Since we repeat $\log n$ times, this turns into 'with high probability.'

Student Question. *Why do we repeat* $\log n$ *times specifically — why not* $\log \log n$*?*

Answer. With $\log n$ we get that the probability we fail every time is $(1 - \frac{1}{2e})^{\log n}$, so we get something that's polynomial in n. That's how we define 'high probability' — we say something is high probability if the failure probability is at most $\frac{1}{\mathsf{poly}(n)}$. (You could define it differently — if you did $\log \log n$, you'd get $\mathsf{polylog}$ instead of polynomial.) The reason is because if we want to do thsi n times, we can union

bound and still get something small. (Here we're union-bounding over pairs (i, j), so we want $1/\mathsf{poly}(n)$ so that by adjusting the constant, we can union-bound and still get 'high probability.')

Finally, what's the runtime of this algorithm? We added one log factor in the unique witness finding; then we added a second log factor in looking for the right value of s; and we added a third log factor in repeating $\log n$ times. So our total runtime is $O(T(n)\log^3 n)$.

§7 February 27, 2025 — Subgraph isomorphism

Today we'll talk about a different problem, subgraph isomorphism — where we want to find if a certain graph exists in a different graph.

Problem 7.1

Given graphs $G = (V, E)$ and $H = (V_H, E_H)$, determine whether G contains a copy of H.

This means we want to have a function from the vertices of H to the vertices of G such that the edges are maintained — we want a one-to-one mapping $f : V_H \to V$ which satisfies some properties. We'll talk about two different versions of this — the *induced* and *non-induced* case.

In the induced case, we want a copy of H which is *exactly* the same — if there's an edge in H there should be an edge in G, and if there isn't an edge in H there shouldn't be one in G. So we require that $(f(u), f(v)) \in E$ if and only if $(u, v) \in E$.

If we don't care about when H doesn't have edges, we call this the non-induced case — we only require one direction of the 'if and only if,' that if $(u, v) \in E$ then $(f(u), f(v)) \in E$. (So here you have less restrictions.)

In general, this problem is NP-complete, even for cliques — if the size of H is part of the input, we can't be polynomial-time. So we'll only talk about the case where H is of *constant* size — where $|V_H| = k = O(1)$.

§7.1 Brute force

What's the brute-force way of finding H in G? For every k-tuple of vertices $(v_1, \ldots, v_k) \in V^k$, we can check whether this is a copy of H — we can name $V_H = \{h_1, \ldots, h_k\}$ and check whether the function $f : h_i \mapsto v_i$ satisfies the condition we want.

How long does this take? We're going over all n^k k-tuples. Then to check these constraints, in the induced case we have to check exactly k^2; in the non-induced case it's the number of edges in H. So this is $O(n^k k^2) = O(n^k)$ time (since k is constant).

Our goal is to beat this runtime, at least for specific graphs — to see if we can detect copies of H more efficiently.

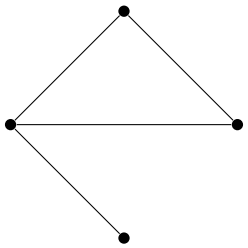
§7.2 Induced vs. non-induced

First, we'd expect the non-induced case to be easier, since there's less restrictions. We'll first show that we can reduce from the non-induced to the induced case — if we can find induced subgraphs, we can also find non-induced subgraphs in the same runtime.

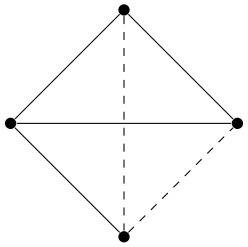
Theorem 7.2

Suppose we have an $O(n^c)$ time algorithm for the induced subgraph problem. Then there is an $\widetilde{O}(n^c)$ time algorithm for the non-induced subgraph problem.

Proof. For concreteness, imagine that H is the following graph.



We know how to find an *induced* copy of this graph. And now we want to be able to find *non-induced* copies, where we don't care if there are extra edges.



So we want to take our graph G and construct some other graph where induced detection on that graph would give us non-induced detection on the original.

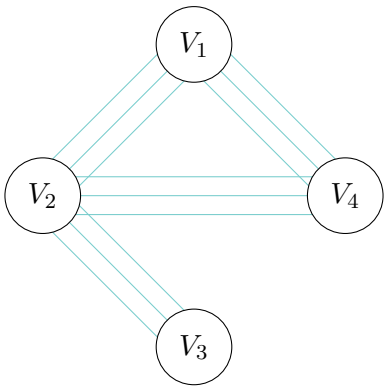
> **Goal 7.3.** Given G, construct G' such that an induced copy of H in G' corresponds to a non-induced copy of H in G.

(We'll actually have some probability here — there'll be an existence claim.)

The idea is we're going to try to delete some edges — we'll find a way to partition the graph such that with some probability, the extra edges get deleted. This is called *color-coding* — we'll assign a color $c \in \{1, \ldots, k\}$ to every vertex in G. We'll do this uniformly at random (and independently for each vertex); we write $c(v)$ for the color of v.

In this case, we have four colors; and then we're going to draw our different vertex sets. So V_i is all the vertices that were assigned color i.

Now, we're not going to draw any edges within vertices of the same color. Meanwhile, between two vertices of colors 1 and 2, we'll add an edge only if they already had an edge, and H had an edge between those two. So here we'll add edges between V_1 and V_2 any time there was an edge in G. But between V_1 and V_3, or V_3 and V_4, we won't add any edges, regardless of whether there were edges in the original graph.



Let's see how this graph helps us. We want an induced copy in this new graph to be a non-induced copy in the original. Now if we find an induced copy of H in this graph, what does that mean about the original graph? We only deleted edges — we didn't add any — so if we find an induced copy of H in G', then all those edges still exist in G, which guarantees there's a non-induced copy of H in G.

What about the other direction? If you have a non-induced copy of H in G, will we find it in the new graph? What's one case when we *will* find it? Say we have a mapping $f : V_H \to V$ which gives a non-induced version of H in G (sending $h_i \mapsto v_i$). If v_1 happened to fall in V_1, v_2 in V_2, v_3 in V_3, and v_4 in V_4, then this is going to be an induced copy of H in G' — v_1 and v_2 were in our original non-induced copy of H, so we know there's an edge between them (we might als ohave additional edges like (v_1, v_3), but the edges in H certainly exist). And these edges survive in the new graph, while edges like (v_1, v_3) don't survive — because we didn't put any edges between V_1 and V_3.

So if for every i, the color assigned to v_i is i, then we'll get an induced copy of H.

What's the probability that this thing happens (for a specific instance of H)? It's k^{-k} — these are all independent, and each vertex has the right color with probability k. This might *look* tiny, but we're assuming k is constant, so it's actually constant!

So now how do we get this to be 'with high probability'? We do this $k^k \log n$ times. So our algorithm is to construct G' and run induced detection, and just repeat this $O(k^k \log n)$ times. Then with high probability, using an algorithm for induced detection, just by adding a log factor (and this big constant) we can get an algorithm for induced detection. □

Remark 7.4. We did this with randomness, but this technique of color coding is actually used a lot, and there are ways to derandomize it. So this can actually be a *deterministic* reduction.

§7.3 Pattern difficulties

What do you think are the hardest patterns to find? Are all the same, or are some easier or harder? Well, if H is just k vertices with no edges, then non-induced detection is trivial. So maybe the more edges we have, the harder it is to find. In the *induced* case, cliques are actually the hardest thing to find. This is a theorem we'll prove on the next problem set:

Theorem 7.5

Suppose there is an $O(n^c)$ time algorithm to detect a k-clique. Then there is an $O(n^c)$ time algorithm to detect an induced copy of *any* k-vertex subgraph.

So this shows cliques are the hardest — if we can detect cliques, then we can detect *any* other subgraph. And the non-induced case actually turns out to be easier in some cases.

To prove this, you'll do something similar to this idea of separating the nodes into different sets by blowing the graph up into some kn-node k-partite graph.

So cliques are the hardest; now we'll talk about how to detect cliques.

§7.4 Triangle detection

Question 7.6. How do we detect a 3-clique (i.e., a triangle).

We could get an $O(n^3)$ brute force algorithm. But we can do better using matrix multiplication — if A is the adjacency matrix of the graph, then A^2 gives us all the 2-paths. So if we consider $A^2 \wedge A$, we'll get a 1 for every pair that has a 2-path and an edge, i.e., a triangle.

We'll talk more about triangle detection and its relationship to BMM — it turns out it's not just this direction, but that they're actually *equivalent*.

So we can do triangle detection in $O(n^\omega)$ time.

§7.5 Larger cliques

What about more general cliques? We're going to want to split based on what k is mod 3 — we'll want to deal with the case where k is divisible by 3. If we could do this, how could we deal with k which *isn't* divisible by 3 (i.e., $k = 3\ell + r$)? So for example, we want to find a $(3\ell + 1)$-clique using some algorithm for a 3ℓ-clique.

To find a k-clique, we need to find a $(k-1)$-clique in the neighborhood of some vertex. So if $r = 1$, then for every vertex, we can run $(k-1)$-clique detection in its neighborhood.

What about the case where $r = 2$? We could just run the $(3\ell+1)$-clique finding algorithm in the neighborhood of every vertex. Or we could take every edge and look at its joint neighborhood, and find a 3ℓ-clique there.

So if $r = 1$ or 2, then in additional runtime n^r, we can reduce to (3ℓ)-clique detection. This means

$$T(3\ell + r) = n^r \cdot T(3\ell).$$

So now we can focus on the case of cliques divisible by 3.

Theorem 7.7 (Nešetril–Poljak)
If k is divisible by 3, then we can detect a k-clique in time $O(n^{k\omega/3})$.

So we can detect a k-clique in the runtime of triangle detection on a graph of size $n^{k/3}$.

Proof. We'll do this by reducing k-clique detection to the problem of triangle detection. How do we do that — how do we construct a graph where a triangle will be equivalent to a clique in the original graph?

Goal 7.8. Construct a graph G' where triangles in G' correspond to k-cliques in G.

This graph is going to be really big (which is where the runtime comes from). We'll have vertices corresponding to $(k/3)$-tuples in G — so for all $v_1, \ldots, v_{k/3} \in V$, we check whether they form a $(k/3)$-clique in G; and if so, we put the tuple $(v_1, \ldots, v_{k/3})$ as a vertex in G.

And our edges will be two tuples who have all edges between them present — for every pair of vertices $(v_1, \ldots, v_{k/3})$ and $(u_1, \ldots, u_{k/3})$, we'll draw an edge between them if they form a $(2k/3)$-clique in G (this means they have to be disjoint, and all edges between them have to be present).

What happens if we have a triangle in G'? So we have three tuples $(v_1, \ldots, v_{k/3})$, $(u_1, \ldots, u_{k/3})$, and $(x_1, \ldots, x_{k/3})$ which are all connected in G'. First, these have to be k different vertices — we can't have any overlap between them (because e.g., if there was overlap between $(v_1, \ldots, v_{k/3})$ and $(x_1, \ldots, x_{k/3})$, we wouldn't have drawn an edge between them; and within a set there can't be overlap either). And we know all the edges exist — all the edges within the u's, v's, and x's exist because they're even vertices in G', and all the edges between them exist because these edges exist in G'. So not only are they k distinct vertices, but they actually form a k-clique. So any triangle in G' actually gives a k-clique in G.

And what about a k-clique in G — does that give us a triangle in G'? If $(v_1, \ldots, v_k)$ is a k-clique in G, how do I get from this to a triangle in G'? We can just divide it into three parts — $(v_1, \ldots, v_{k/3})$, $(v_{k/3+1}, \ldots, v_{2k/3})$,

and $(v_{2k/3+1}, \ldots, v_k)$ are each going to be a vertex, and all the edges between them are going to exist. So this is a triangle in G'.

So G' has size up to $O(n^{k/3})$, and G' has a triangle if and only if G has a k-clique. So building G' and running triangle detection on G' will give us a k-clique in our original graph.

And what's the runtime of doing this? First, how long does it take to construct G'? We have to check every $(k/3)$-tuple in G to check whether it's a $(k/3)$-clique. And then we have to check at most every $(2k/3)$-tuple to check whether it should be an edge. So constructing G' takes at most $n^{2k/3}$ time. And we also run triangle detection on G', which takes time $O((n^{k/3})^\omega) = O(n^{k\omega/3})$. This is bigger than $O(n^{2k/3})$, so it dominates our runtime. □

Remark 7.9. This is actually the best-known way to detect cliques; it's one of the weak fine-grained hypotheses (that there's no better way). It's a weak conjecture, so maybe it'll be proven wrong and there is a better way to detect cliques. But basically the best way we know how to detect cliques is just to detect triangles in very big graphs.

Student Question. *If there's a triangle, there'll be lots of triangles, right, since there's lots of ways to partition a k-clique into three parts?*

Answer. Yes, because every clique will give us lots of triangles. But it's unclear how to use this, since we're only working with detection. (This is the best-known bound even for randomized algorithms.)

§7.6 Induced detection of other graphs

For cliques, induced and non-induced detection is the same, because we have all the edges. But now we'll start separating based on induced vs. non-induced.

In the non-induced case, lots of graphs are much easier than cliques. We already mentioned k-independent sets — if we have k isolated vertices and want non-induced detection, we can do this in *constant* time, because all we have to do is check whether we have at least k vertices.

There's lots of other k-sized graphs that are actually easy to detect. If we want to find a k-path, we can do that in linear time, i.e., $O(n+m)$. If we want to find a k-cycle where k is even, this takes $O(n^2)$ time. If we want a k-cycle where k is odd, then k-cycle detection is actually equivalent to triangle detection (we'll see this later), so we can do this in $O(n^\omega)$ time.

So we have all these non-induced graphs which are easier to detect than cliques. But what about the induced versions?

For k-independent sets, this is the same as a clique (you just switch edges and non-edges).

What about the others? Is any induced graph the same as detecting a clique, because for every edge we're requiring whether it exists or not? That's not immediately obvious. But we'll show that for specific values of k, *anything* except the k-clique and k-independent set is actually easier to detect (in the induced case).

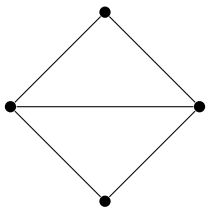
Theorem 7.10 (Vassilevska Williams–Wang–Williams)

Let H be any 4-node graph which is not the 4-clique or 4-independent set. Then we can detect if there's an induced H in time $O(n^\omega)$.

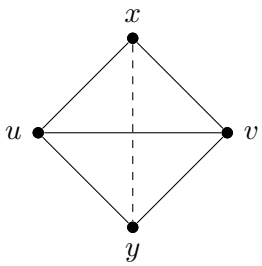
And n^ω is the time it takes for us to detect a 3-clique. For $k = 5$ and 6, the same result was shown — if $|V_H| = k$ and H is not a k-clique or k-independent set, then induced H can be detected in the time it takes to detect a $(k-1)$-clique.

For general k, or anything bigger than 6, this is an open problem — we don't know if this is true. But for k up to 6, there's this strict split where clique and independent set are hard, but anything else can be done faster. (And for $k = 5$, some patterns are even easier — e.g., they can be done in $O(n^\omega)$.)

We're not going to show the entire theorem; but we'll consider one specific H, which gives an idea for how this was proven. We'll look at the subgraph of size 4 called a *diamond* — a clique minus one edge. We'll show that detecting an induced H can be done in n^ω time.



How do we find a graph like this? If we have some edge uv and we know all the triangles on top of it, any pair of triangles sharing the edge uv would form either a H or a clique (if they have an edge between them they'd form a clique, and otherwise they form a diamond).



So how would we try to count these? First, how do we count the number of triangles that share this edge uv? When we detected triangles before, we did matrix multiplication to see how many 2-paths there were from u to v — if (u, v) is an edge, then

$$A^2[u, v] = \#\text{triangles on } (u, v).$$

So how do I get the number of pairs (x, y) such that both form a triangle on uv? We can just take this quantity choose 2 — so

$$\binom{A^2[u, v]}{2} = \#\text{pairs } (x, y).$$

And each of these pairs either forms a clique or a diamond — so this is equal to the number of diamonds (of the above form, where u and v are the middle), plus the number of 4-cliques.

Now what if I sum over all the edges? We're going to get

$$\Gamma = \sum_{(u,v)\in E} \binom{A^2[u, v]}{2} = \#\text{diamonds} + 6 \cdot \#\text{4-cliques}.$$

(Each diamond is only going to get counted once — there's only one choice of (u, v), but each clique is going to get counted once for each edge, and there's 6 edges.)

And that's the key thing — we have some separation between the diamonds and cliques, because the cliques get counted more than the diamonds.

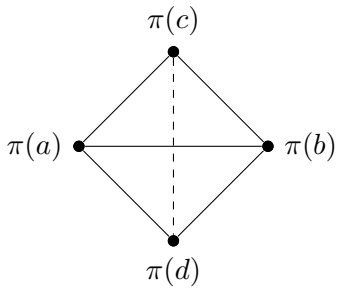
Are we done here? We want to know whether we have a diamond in the graph or not. So we run this and compute this sum. If we get 6, do we know if that means there's 6 diamonds or 1 clique? No. If we get

something that's not 0 mod 6, then we know there's a diamond (because the cliques contribute 0 mod 6). But if we get something that *is* 0 mod 6, then we're not done.

The idea is to try to subsample some part of the graph — if we have an original graph where the number of diamonds is 0 mod 6, we want to argue that if we randomly take some part of the graph, then we're going to keep some number of diamonds, and the probability that number is also 0 mod 6 is going to be low (or constant — something we can amplify to get it to be very low). So we're going to randomly sample and hope that the number of diamonds there is not 0 mod 6 (assuming the number of diamonds was not 0 in the original graph).

Our sampling is going to be very simple — we just keep every vertex with probability 1/2. And now the argument here is going to be a little more complicated, but in the end, we're only going to argue about the probability here (the algorithm is still very simple — we just keep every vertex with probability 1/2, and on the sub-sampled graph, we run this algorithm where we compute Γ as above).

Let's label the vertices of G by 1, 2, ..., n. We'll say that four vertices $a, b, c, d \in V$ *induce a diamond* if we can permute them so that they form an (induced) diamond, meaning that these permuted vertices have exactly the edges we want.



Now to count the number of diamonds in the graph, we'll define a polynomial in n variables $x_1, \ldots, x_n$ — we're going to sum over all four indices which induce a diamond, and for every such 4-tuple, we're going to add the corresponding monomial. So we define

$$P(x_1, \ldots, x_n) = \sum_{\substack{i_1<i_2<i_3<i_4 \\ (i_1,i_2,i_3,i_4) \text{ induces a diamond}}} x_{i_1}x_{i_2}x_{i_3}x_{i_4}.$$

What is this saying? If we think of the x_i's as taking values in $\{0, 1\}$, for every diamond, we'll count it if its vertices are all 1, and otherwise we'll get 0.

What is P if we just put 1 into all these variables? Then it's the number of diamonds.

But why did we do this? For a subset $S \subseteq [n]$, we'll define its indicator vector as

$$\chi_S = \begin{cases} x_i = 1 & \text{if } i \in S \\ x_i = 0 & \text{if } i \notin S. \end{cases}$$

Then $P(\chi_S)$ is the number of diamonds in $G[S]$ (our graph G restricted to S, where we take G and only keep the vertices in S) — because for every diamond, we only get a 1 here if $x_{i_1} = \cdots = x_{i_4} = 1$, which happens when all four of $i_1, \ldots, i_4$ happen to be in S.

So this polynomial helps us count how many diamonds are left after we sample the graph.

We don't actually know how to construct this polynomial — this is just an argument for the probability. (Constructing this polynomial would require us to find *all* the diamonds.)

On problem set 2, we showed (or will show) a useful lemma:

Lemma 7.11

For any $m \geq 2$, if $P(x_1, \ldots, x_n)$ is a nonzero multilinear polynomial over $\mathbb{Z}_m$ of degree d, then

$$\mathbb{P}_{(a_1,\ldots,a_n)\in\{0,1\}^n}[P(a_1, \ldots, a_n) \neq 0] \geq \frac{1}{2^d}.$$

Definition 7.12. A *multilinear polynomial* is a polynomial where no variable is raised to a power higher than 1, i.e., a polynomial of the form

$$P(x_1, \ldots, x_n) = \sum_{S\subseteq[n]} c_S \prod_{i\in S} x_i.$$

Its *degree* is the largest size of a set S with nonzero coefficient c_S.

So if d is constant, then for any nonzero polynomial, the probability it evaluates to something nonzero mod m is some constant.

How do we use this to solve our problem for diamonds? Our polynomial P was a sum over diamonds, so it's a degree-4 multilinear polynomial. And we care about P being 0 mod 6, so we're evaluating over $\mathbb{Z}_6$.

What else do we need? We need P to be nonzero; and here we know that's the case because we're assuming there *is* an induced diamond.

So now P is exactly in the setting we can plug into this lemma; and this tells us that

$$\mathbb{P}_{a\in\{0,1\}^n}[P(a) \not\equiv 0 \pmod 6] \geq \frac{1}{2^4}.$$

And what this means is that the number of diamonds in the subsampled graph is nonzero mod 6. So what the lemma gives us is that if we sample a random subset S of the graph, then

$$\#\text{diamonds in } G[S] = P(\chi_S),$$

which means that

$$\mathbb{P}[\#\text{diamonds in } G[S] \not\equiv 0 \pmod 6] = \mathbb{P}_{\chi_S\in\{0,1\}^n}[P(\chi_S) \not\equiv 0 \pmod 6] = \Omega(1).$$

So we had an issue where our number of diamonds was 0 mod 6, and this lemma tells us that sampling is going to fix that issue with constant probability. Now we just have to repeat.

So to detect a diamond in the graph, we're repeatedly going to sample S where we include every vertex with probability $1/2$, and consider the new graph $G' = G[S]$. And then we take A' to be the adjacency matrix of G'; we compute $(A')^2$, and then we compute this sum

$$\Gamma' = \sum_{(u,v)\in E(G')} \binom{(A')^2[u,v]}{2} \pmod 6.$$

If this is not 0 mod 6, then we've found a diamond. If it is 0 mod 6, we just do this again — we repeat this $O(2^4 \log n)$ times.

We've already said this, but let's write down correctness and runtime. For correctness, if there is a diamond, then P is not 0, so the lemma tells us that we'll return YES with high probability (at some point we'll get something nonzero mod 6). And if we return YES, there's always a diamond — we're never wrong about that.

And what's the runtime of this algorithm? Note that this polynomial isn't part of the algorithm — we never built it. So the runtime is just doing this calculation. And the expensive step is computing $(A')^2$ — everything else we can do in $O(n^2)$ time. So we get n^ω; but we're repeating $\log n$ times, so we get $\widetilde{O}(n^\omega)$.

This is the same idea that's applied to every 4-node graph. There's some point where it's not completely symmetric like a k-clique, so that if we sum over some way of counting it, then we get some count of the graph we're actually looking for, and some count of cliques times a number. So if we only look at our sum mod that number, then if it's nonzero we know our graph exists. And we can use the same subsampling lemma.

Student Question. *Can we count the number of diamonds?*

Answer. This shouldn't let you count the number of diamonds. Counting cliques is harder than just detecting cliques; Yael isn't sure if there's specific work on counting subgraphs that aren't cliques.

§8 March 4, 2025 — Minimum weight triangles

Today we'll talk about finding triangles in a graph of *minimum weight*. You might ask, why triangles? We always focus on them because it turns out if you can find a triangle fast, you can find any pattern you want — most algorithms for pattern detection in graphs rely on a reduction to triangle detection. And we often want to find not just a pattern, but a best possible one. We usually encode this by putting weights on the edges or vertices (sometimes both), and we want a pattern of minimum weight.

Usually you can take any one of your favorite patterns (e.g., a k-clique) and reduce to the problem of finding a triangle in a slightly larger graph. So today we'll study triangles. We'll study two variants — one where the weights are on the edges, and one where they're on the nodes. These are actually very different. When there are weights on vertices, we'll see there are algorithms that are basically as good as the unweighted case. But when there's weights on the edges, we'll see the problem becomes equivalent to APSP, which is much harder.

Student Question. *Can you reduce vertex weights to edge weights?*

Answer. Yes — for every edge, you take the average of the weights of its two endpoints, and then the min edge-weight triangle corresponds to the min vertex-weight triangle. You can also see that if you have weights on both the vertices and edges, you can reduce to just edge weights (where you take the vertex weights and add them onto the edge weights, appropriately scaled).

But it could have been you could reduce the edge-weighted case to the vertex-weighted case — maybe it's possible and we don't know how to do it. But as far as our current knowledge goes, the node-weighted case is equivalent to unweighted triangle detection, and the edge-weighted case is equivalent to APSP.

Problem 8.1

- **Input:** a graph $G = (V, E)$ and weights $w : E \to \mathbb{Z}$ or $w : V \to \mathbb{Z}$.
- **Output:** a triangle (a, b, c) in G with $w(a, b) + w(b, c) + w(a, c)$ (if we have edge weights) or $w(a) + w(b) + w(c)$ (if we have vertex weights) minimized (or No if there are no triangles).

As usual, with triangle problems, because we only care about triangles, you can always assume your graph is tripartite. Usually the way you do this is by taking every vertex from the original set and creating three copies. So now you have three sets (corresponding to the three copies) which are independent sets; every triangle appears 6 times, and you don't create any new triangles. So for any triangle problem — not just min-weight triangles, but any version — we can assume without loss of generality that the graph is tripartite.

Let's call the parts A, B, and C; then we're actually just looking for $a \in A$, $b \in B$, and $c \in C$ such that (a, b, c) is a triangle (meaning every one of these edges appear), and the sum of weights on the edges is minimized (or the sum of weights on the vertices, in the vertex-weighted case).

Another thing we'll assume is that the weights are integers. In fact, for the node-weighted case that won't matter (they won't need to be integers). But our proof for the edge-weighted case will rely on that; we'll assume they lie in the range $\{-M, \ldots, M\}$. We want M to have $O(\log n)$ bits (it could have more, but not polynomial in n).

§8.1 Min node-weighted triangle — a warmup

What we're going to do first is take the node-weighted version, and slowly develop algorithms for it.

As a warmup, we're going to create a subcubic algorithm (and then we'll get an algorithm that's roughly n^ω time). Specifically, we'll see an $O(n^{2+1/(4-\omega)})$ time algorithm.

> **Remark 8.2.** Where did we see this runtime before? It's Zwick's runtime for unweighted directed APSP. That algorithm was somewhat involved. But it turns out you can get the same runtime with a very simple algorithm for this problem. (But Zwick is for unweighted graphs, and what we have here is a weighted problem, where there are arbitrary weights on the nodes.)

So we're given a graph with parts A, B, and C, and we have some weights on the nodes. Instead of just trying to solve 'does there exist a triangle,' let's try to solve the problem:

> **Goal 8.3.** For every edge $(a, b) \in A \times B$, find the minimum-weight triangle using that edge — i.e.,
>
> $$\min\{w(a) + w(b) + w(c) \mid c \in C, (a, c), (b, c) \in E\}.$$

Once we have the minimum weight of a triangle involving each particular edge, I can go through all the edges and find the minimum (and this will find me the minimum-weight triangle in the whole graph).

Because I'm doing this for every edge separately, once I fix a and b, the part $w(a) + w(b)$ can be moved outside the minimum — so this is actually

$$w(a) + w(b) + \min_{c:(a,c),(b,c)\in E} w(c).$$

And the point is since I've split $w(a)$ and $w(b)$ off, all of a sudden even $w(c)$ doesn't matter too much.

So what we'll do is sort C by weight — we sort $c \in C$ by $w(c)$ (so c is before c' if $w(c) < w(c')$). Now this becomes a problem about finding the minimum $c \in C$ such that (a, c) and (b, c) are edges. So I want to find the minimum vertex in the graph, when sorted by weight, so that there's an edge from a to c and from c to b.

So I've sorted C in order of weight; and now I want to find, for every *fixed* a and b, the first c that has a path of length 2.

Now, because I fixed a and b, I can forget about $w(a) + w(b)$. And this becomes sort of a BMM question, except that instead of checking whether there *exists* c such that (a, c) and (b, c) are edges, I'm asking what's the smallest? This is called the *min witness product*.

§8.2 Min witness product

Problem 8.4 (Min witness product)

- **Input:** two $n \times n$ Boolean matrices A and B.
- **Output:** the matrix $C = A \odot B$ defined by

$$C_{ij} = \min\{k \in [n] \mid A_{ik} = B_{kj} = 1\}$$

(or ∞ if there is no k).

So you're outputting some matrix of integers where for every (i, j), I output the minimum witness — the minimum k where A_{ik} and B_{kj} are both 1. This is very similar to BMM, and the first time you see it, you might think, why is it not in n^ω time? Maybe it is. But the fastest known algorithm for this is something we'll see soon (and its runtime is what we said; you can do a bit better using rectangular matrix multiplication if $\omega > 2$, in which case you can get $n^{2.53}$).

So now we'll see how to solve this problem; if we can do it, then we can find for each edge what's the minimum weight triangle containing it, and then we can solve the original problem.

Claim 8.5 — Min witness product is in $O(n^{2+1/(4-\omega)})$ time.

Proof. This is much easier than many of the other matrix products we've seen before. Suppose we have our matrix A and B (which are Boolean). And we want to find a minimum over k's. So what we're going to do is split the columns of A and the corresponding rows of B into buckets. (This is all we know how to do; it turns out we don't know how to do better.) So we'll split the columns of A and rows of B into buckets of size g (so there's n/g of them), which we call $A_1, \ldots, A_{n/g}$ and $B_1, \ldots, B_{n/g}$. Then we compute the Boolean products of the corresponding slices — i.e., $A_i \cdot B_i$ for each $i \in [n/g]$.

Each multiplication is an $(n \times g)$ by $(g \times n)$ matrix product. What's the runtime for that? What we do is the standard reduction from rectangular to square — we split both of them up into $g \times g$ blocks. And now we have to compute $(n/g)^2$ pieces of size $g \times g$; so this is

$$\left(\frac{n}{g}\right)^2 \cdot g^\omega.$$

(There are faster algorithms that don't do blocking like this — for each particular tiny dimension in terms of n, there are faster ways than reducing to square matrix multiplication. This is where you get the $n^{2.53}$ bounds. But today we'll just do this.) So you chunk into $g \times g$ blocks and do $(n/g)^2$ matrix multiplications; and the total runtime is

$$\frac{n^3}{g^{3-\omega}}.$$

What do we get from this? Instead of doing an $n \times n$ matrix multiplication, we're doing something more expensive — for each bucket, we compute a matrix product.

But what do we want? We want that for each (a, b), we want the minimum c such that there's an edge. And we know for each a and b, the smallest bucket for which there is a 1. So what we've learned from here is for every a and b, we know the minimum i such that $(A_i \cdot B_i)_{ab} = 1$. And the minimum witness for a and b is going to be in the minimum bucket (because these things are sorted). We're looking for the smallest place where there's a 1 between a and something in this bucket, and that same bucket and b; and that guy is in the minimum i.

And now for each (a, b), we'll look at that particular i, and try each one of the elements in this bucket to find the minimum. (This sounds kind of stupid, but it's the best thing we know how to do.) So for every (a, b), if this minimum i is $i(a, b)$, then we try every c in the $i(a, b)$th bucket — this is just brute force.

But now instead of computing the brute force for $n^2 \cdot n$, it's $n^2 \cdot g$ — we've narrowed our search to some small bucket, instead of the whole bucket.

So we used matrix multiplication to figure out for every (a, b), some small range of the possible witnesses. Each (a, b) could have a different range. And then we just brute force in there.

> **Remark 8.6.** You might wonder, why not recurse? But we don't see how to recurse. The problem is that there's n^2 choices of (a, b), and you could have a bunch of completely uncorrelated ones corresponding to each bucket. So that doesn't correspond to a nice matrix product thing — they don't distribute nicely.

So we pay for having to run brute force, and the final runtime is

$$n^2 g + \frac{n^3}{g^{3-\omega}}.$$

Then you end up setting $g = n^{1/(4-\omega)}$, and you get the runtime of $O(n^{2+1/(4-\omega)})$ that we promised. (It happens to be exactly the same runtime as in Zwick's algorithm because the bottleneck case of Zwick has the same sort of tradeoff.) □

This min-witness product is the simplest out of these matrix products for which we don't have an n^ω time algorithm — even if ω is 2 the best we have is $n^{2.5}$, and any sort of improvement would be very interesting.

§8.3 An $\widetilde{O}(n^\omega)$ time algorithm for min node-weight triangle

What we're going to do now is note that this was overkill. We computed for *every edge* what the minimum weight triangle is. But it turns out if we just want the minimum weight triangle in the entire graph, we don't need to compute this for every edge; and it turns out we can have a faster algorithm (essentially n^ω).

This time, instead of reducing to some bizarre matrix product, we're just going to reduce to triangle detection.

Assume we have A, B, and C, and all of the parts are sorted by weight.

As is customary, we're going to pick some parameter g (you can think of it as $g \sim \log n$ — so it's small). And unsurprisingly, we're going to split the vertex set into pieces (or really, each of A, B, and C); but this time the pieces are going to have size n/g.

So we have $A_1, \ldots, A_g$; $B_1, \ldots, B_g$; and $C_1, \ldots, C_g$.

Because g is so tiny, I can afford to take every triple of these buckets and run triangle detection on it, disregarding all the weights. So for all $a, b, c \in [g]$, we run unweighted triangle detction — which we can do in $O((n/g)^\omega)$ time — on the subgraph induced by $A_a \cup B_b \cup C_c$. So we pick some triple of parts and focus on this subgraph, and just run triangle detection, disregarding the weights.

We then form a list

$$L = \{(g_1, g_2, g_3) \in [g]^3 \mid A_{g_1} \cup B_{g_2} \cup C_{g_3} \text{ contains a triangle}\}.$$

So now you form a list of these triples of indices so that the corresponding triple of pieces contains a triangle. And the size of this list is g^3.

Now a very important property is that we only care about the minimum weight triangle. This means if (g_1, g_2, g_3) and (g_1', g_2', g_3') are in L and $g_1 < g_1'$, $g_2 < g_2'$, and $g_3 < g_3'$, what can we do? So our second triple (g_1', g_2', g_3') is bigger than the first in every coordinate. Do we need to consider it? No! Because we sorted the vertices by weight. So if I found a triangle among the yellow triple A_{g_1}, B_{g_2}, and C_{g_3}, and then I found

a triangle in some later green triple, then the triple in the yellow triangle is strictly better than the one in the green triple.

So if one triple dominates another in every coordinate, then the second can be ignored — we can remove (g_1', g_2', g_3') from L.

So we take this list, and we compare every pair of triples; and we kill off a triple if it's dominated by something also in L. This takes g^6 time; g is logarithmic, so we don't care.

Now our list contains only undominated triples. And what do we do? We recurse on every one — so we recurse on all $A_a \cup B_b \cup C_c$ with $(a, b, c) \in L$. There's some base case; if $n \leq g$ then we do brute force in $O(g^3)$ time. (In this case we can't split into multiple buckets, so we brute force there. But g is tiny — it's the log of the *original* n.)

What's the runtime recurrence? In terms of the size of L (which we'll discuss later), it's

$$T(n) \leq |L| \cdot T\left(\frac{n}{g}\right) + O\left(\left(\frac{n}{g}\right)^{\omega} \cdot g^3 + g^6\right).$$

We assumed g is small, so the g^6 is dominated by the first term and we can ignore it.

So really in order to analyze the runtime, we care about what is $|L|$. And we'll prove that L is actually small.

> **Claim 8.7 —** If L is a list of triples from $[g]^3$ such that there do not exist (g_1, g_2, g_3) and (g_1', g_2', g_3') with $g_i < g_i'$ for all $i \in [3]$, then $|L| \leq 3g^2$.

So not only is this subcubic in g — it doesn't contain g^3 triples — but it actually contains quadratically many.

If this is true, then what we'll get is

$$T(n) \leq 3g^2 T(n/g) + n^{\omega} g^{3-\omega}$$

(up to constants). And then it's basically an exercise that if you set $g = \log n$, this will become $n^{\omega + o(1)}$. (What you end up doing is you expand the recursion over and over, and then you see that any dependence on g becomes $n^{o(1)}$.)

So the important part is to prove this claim.

Proof. We're going to give $3g^2$ *chains* of triples that are going to cover the entire space $[g]^3$.

Consider the following: for all $g_1, g_2 \in [g]$, we consider the chains

$$(1, g_1, g_2) < (2, g_1 + 1, g_2 + 1) < \cdots < (1 + i, g_1 + i, g_2 + i) < \cdots$$

(so you start with $(1, g_1, g_2)$, and you add some integer i to each of the coordinates — this is a chain). And you do the same with $(g_1, 1, g_2)$ and $(g_1, g_2, 1)$ — so we also get a chain

$$(g_1, 1, g_2) < (g_1 + 1, 2, g_2 + 1) < \cdots < (g_1 + i, 1 + i, g_2 + i) < \cdots$$

and similarly with $(g_1, g_2, 1)$.

So for any $g_1, g_2 \in [g]$, we have these three chains.

We claim that any triple (g_1, g_2, g_3) appears in one of these chains. Why is this true? You can subtraact out the minimum — if g_2 is the minimum, then you let $i = g_2 - 1$, and then this will define which pair $g_1, g_2 \in [g]$ we should consider and go there on the respective chain.

So these chains cover all the triples. And L, because it doesn't contain any two triples with one dominating the other, basically L contains at most one element from each triple — because they're chains. So therefore $|L|$ is at most the number of chains, which is $3g^2$. □

So now we have the bound for our runtime recurrence; and you expand it out, and you get $n^{\omega+o(1)}$.

Let's see the computation. So we've shown that

$$T(n) \leq 3g^2 T(n/g) + n^\omega g^{3-\omega}.$$

When we expand out, we're going to have

$$T(n) \leq \sum_{i=1}^{\log_g n} (3g^2)^i \left(\frac{n}{g^i}\right)^\omega g^3.$$

Some of this stuff goes in front — the g^3 is going to go in front, and the n^ω goes in front — and we get

$$n^\omega g^3 \sum_{i=1}^{\log_g n} 3^i g^{i(2-\omega)}.$$

And $\omega \geq 2$, so this is bounded by

$$\sum_{i=1}^{\log_g n} 3^i \leq O(3^{\log_g n}).$$

And because $g \sim \log n$, this is

$$O(3^{\log n/\log\log n}) = n^{o(1)}.$$

So we get $n^\omega g^3 n^{o(1)}$; and $g \sim \log n$, so g^3 is $\mathsf{polylog}(n)$.

So up to $n^{o(1)}$ factors, this problem is basically the same as triangle detection with no weights.

> **Remark 8.8.** This is not the only way to get an algorithm with this runtime for min-weight triangle — there's another way as well — but that also ends up paying an $n^{o(1)}$. But it's still roughly n^ω.

> **Remark 8.9.** We'll point out a few things. This can be viewed as a reduction to triangle detection — so if you had some other algorithm for triangle detection (not based on matrix multiplication), this would still work.
>
> The other thing is, however, because of the $n^{o(1)}$ factor here, if you end up using combinatorial matrix multiplication algorithms, this won't necessarily give you subcubic time. If you play with g you might get something, but it's not obvious. For example, four-Russians won't give you something subcubic. (Maybe with the current best algorithm you can play with g and get something; but Virginia hasn't tried.)

§8.4 Min edge-weighted triangle — a roadmap

Now we'll see that the problem becomes much harder when the weights are on the edges. Our goal will be to show that this problem is equivalent to APSP on graphs with weights on the edges (which are polynomial in n).

We're going to give aset of reductions. We'll start with APSP on n-node graphs and weights of $O(\log n)$ bits. Before, we saw that this problem is equivalent to the $(\min, +)$-product problem, where you're given $n \times n$ integer matrices and you want to compute

$$C_{ij} = \min_k (A_{ik} + B_{kj}).$$

And we know that if $(\min, +)$ is in subcubic time, so is APSP (they have exactly the same complexity, up to constant factors).

What we're going to do next is reduce this to something called *all-pairs negative triangles* (which we'll define soon). Then we're going to reduce this to *negative triangles*, and reduce that to min-weight triangles. And min-weight triangles easily reduces to the $(\min, +)$-product.

So this is a cycle of reductions. We'll first see the reduction from min-weight triangle to $(\min, +)$-product (because it's very simple, analogous to what we did before). Then we'll define these two other problems and talk about the rest of the reductions.

§8.5 Min-weight triangle to min-plus product

Suppose we have our three sets A, B, and C, and we want to find a, b, and c (with edges between them) with minimum $w(a,b) + w(b,c) + w(a,c)$.

First, suppose the weights are in $\{-M, \ldots, M\}$. Then we can make the graph a complete tripartite graph — if there's no edge between some pair of vertices, then we can add an edge and weight it in an appropriate way (without affecting the answer).

Suppose there's no edge between some $a' \in A$ and $b' \in B$. I want to add an edge and put a weight in here. What should that weight be in terms of M? What I want to achieve is when I add the weight on this edge, I don't create any new good triangles — any triangle I create has weight strictly bigger than any original triangle, so it'll never be a minimum-weight one.

So let's add an edge of weight $5M + 1$ (so every non-edge becomes this weight). Then even if I create a triangle, the minimum weight of the rest of the path is $-2M$ (if both of those edges were the smallest weight); so the total weight of this new triangle (which wasn't in the original graph) would be $3M + 1$. And this is bigger than any true triangle (which would have weight at most $3M$). So I can do this without loss of generality — every non-edge becomes some edge with some really big weight, and it doesn't hurt us when we look for min-weight triangles.

So now without loss of generality we have a complete graph.

Now we're actually going to compute the min-plus product of the matrix corresponding to the edges between A and C, times the matrix corresponding to the edges between C and B.

So we define X to be the generalized adjacency matrix of $A \times C$, and Y to be the generalized adjacency matrix of $C \times B$ — in particular, $X_{ac} = w(a,c)$ and $Y_{cb} = w(c,b)$ (the graph is complete bipartite, so these weights always exist). And now we compute the $(\min, +)$-product of X and Y. This will give me

$$(X \star Y)_{ab} = \min_c (w(a,c) + w(c,b)).$$

So for eveyr one of these edges ab, I've computed the minimum-weight path of length 2 going between them. Very similarly to the node-weighted case, we can then go through every pair of vertices and sum up the weight of this edge $w(a,b)$ plus the weight of the 2-path, and we've found a minimum-weight triangle.

So we can actually solve *all-pairs* mininum weight triangle with this algorithm. That's why this reduction is very easy; and it seems like $(\min, +)$-product could actually be harder, because we're solving all-pairs (as with the node-weighted problem, where all-pairs was $n^{2.5}$ but the actual problem had n^ω). But here it turns out there's an equivalence, which is interesting.

§8.6 Negative triangle problems

So we've already shown that min-weight triangle can be reduced to $(\min, +)$-product (or APSP). But now we're going to show that $(\min, +)$ can be reduced to these two intermediate problems which are actually easier than min-weight triangle.

Now let's define these intermediate problems.

Problem 8.10 (Negative triangle)

- **Input:** $G = (A \cup B \cup C, E)$ and weights $w : E \to \mathbb{Z}$ (where edge weights are in $\{-M, \ldots, M\}$).
- **Output:** decide whether there exists $a \in A$, $b \in B$, $c \in C$ such that $w(a,b) + w(b,c) + w(c,a) < 0$.

(Similar to min-weight triangle, we can assume the graph is complete tripartite, so we don't have to write here that (a,b), (b,c), and (c,a) are edges.)

Problem 8.11 (All-pairs negative triangle)

- **Input:** $G = (A \cup B \cup C, E)$ and weights $w : E \to \mathbb{Z}$ (where edge weights are in $\{-M, \ldots, M\}$).
- **Output:** For each $a \in A$ and $b \in B$, decide whether there exists $c \in C$ such that $w(a,b) + w(b,c) + w(c,a) < 0$.

So one question says, is there any triple with a negative weight sum? In the other, we need to know, for every a and b, is theere a negative weight sum?

The reduction from negative triangle to min-weight triangle is very easy — if we have an algorithm for the min-weight triangle, then we can decide if there's a triangle of negative weight (by checking whether the min-weight is negative).

And if you can solve APNT then of course you can solve NT; but we need the reduction in the opposite direction.

So the only thing left is the reduction $\mathsf{APNT} \leq \mathsf{NT}$ and $(\min, +) \leq \mathsf{APNT}$ (the former is the main part of the reduction — where you take something where you have to do all pairs, and reduce to a single existence question; the latter is easy).

§8.7 Min-plus product to APNT

Now we'll show a reduction from $(\min, +)$-product to APNT.

Let's look at $(\min, +)$-product as a graph problem. This means we have a tripartite graph A, B, and C, and we want to know, for every $a \in A$ and $b \in B$, the minimum weight of a path of length 2 between them.

Now, we're going to add edges — let's add a bunch of edges between A and B. And we'll put weights on them — we're going to put new weights $w(a,b)$ on them, cleverly (creating a complete bipartite graph between them).

And now I feed this to APNT, and solve APNT. This tells me, for each $a \in A$ and $b \in B$, does there exist $c \in C$ such that

$$w(a,c) + w(c,b) < -w(a,b)?$$

(This is because it's $w(a,c) + w(c,b) + w(a,b) < 0$, and we've moved $w(a,b)$ to the other side.)

Why did we write it this way? Now we can set $w(a,b)$ so that we can do binary search. Originally, these weights $w(a,b)$ and $w(c,b)$ were in some range $\{-M, \ldots, M\}$; this means their 2-path sums are going to be in some range $\{-2M, \ldots, 2M\}$. And I need to decide for every (a,b), which of these is the minimum. So I can think of each (a,b) as having some interval I know the minimum lies in (which starts at $[-2M, 2M]$).

Then I use APNT to halve the interval for each (a,b) simultaneously. I initially put $w(a,b) = 0$ for each (a,b), and this tells me for which pairs the minimum is in $[-2M, 0]$ vs. $[0, 2M]$. And then I have new intervals, and I do it again.

Each call to APNT allows me to halve the interval *simultaneously* for all pairs. So I can do a binary search simultaneously for each (a,b); and the number of calls is $O(\log M)$.

So simultaneous binary search with $O(\log M)$ calls to APNT can solve $(\min,+)$-product. So whatever running time I have for APNT, I get the same runtime for $(\min,+)$-product up to some $\log M$ factor. (Here we really use the fact that the weights are integers. But it turns out even if they're reals, you can do other tricks to halve the interval, in which case you'll get $\log n$.)

§8.8 All-pairs negative triangle to (single) negative triangle

This part is actually going to have a loss. It won't be that big of a loss — if NT is in $n^{3-\varepsilon}$ time for some ε, we'll get that APNT is in $n^{3-\varepsilon/3}$ time. So there is a loss in the savings over n^3. But for any savings over n^3 for NT, you'd get a saving for APNT, and therefore APSP would be in subcubic time. Since we don't know subcubic algorithms for APSP, this cycle shows us in order to get one, the only thing you need to do is solve negative triangle (or min-weight triangle) in subcubic time.

> **Remark 8.12.** In fact, you don't even need n^ε savings — for example, if you get $n^3/2^{\sqrt{\log n}}$ savings here, you'd also get something like that for APSP. And in fact the best known algorithm for APSP really just solves the min-weight triangle problem.

We're going to try to do this in the next ten minutes; it's actually not hard, but conceptually it's kind of crazy.

We're going to assume I have a negative triangle algorithm that runs in n^c time; and not only does it decide whether there's a negative triangle, but it finds one if one exists. So I assume I have a $n^{3-\varepsilon}$ time NT algorithm that also finds a negative triangle (if one exists). It's an exercise to show that if you can decide whether a graph has a negative triangle, then in essentially the same time, you can also find one. So this assumption is without loss of generality.

Let's redraw our graph with A, B, and C, and the edges between them. We want to find for every pair (a,b), is there a negative triangle that contains that edge?

Unsurprisingly, we chop up the parts A, B, and C into some pieces of size g (the same thing as before); we call these $A_1, \ldots, A_{n/g}$, $B_1, \ldots, B_{n/g}$, $C_1, \ldots, C_{n/g}$.

NOw we have an output matrix, which we'll call D. Originally it's all 0's. In the end, what it should have is that

$$D_{ab} = \begin{cases} 1 & \text{if exists negative triangle with } (a,b) \\ 0 & \text{otherwise.} \end{cases}$$

So originally we don't know anything, and we'll be filling this matrix in as we go along.

The algorithm will be very simple, and clearly correct; what won't be simple is the runtime.

For all $(a,b,c) \in [n/g]^3$:

- While $A_a \cup B_b \cup C_c$ has a negative triangle (which we detect by running the NT algorithm):
  - Let $x \in A_a$, $y \in B_b$, $z \in C_c$ be a negative triangle. So we've learned that right now, xyz is in a negative triangle. Now we've learned this edge xy is in a negative triangle, and we'll never need to consider this edge again — so we set

    $$D_{xy} = 1,$$

    and we remove the edge (x,y) from G. (We remove this edge from the *entire* graph, so whenever we run the algorithm involving a triple that could have contained x and y, it doesn't see that edge.)

And in the end, we return D.

So we take every triple and keep running negative triangle until we no longer see any negative triangles in the triple, and then we move on.

This solves the problem — we do this for every single triple, and every edge is going to be considered with every triple. So if it were in a negative triangle, we'd find it.

So correctness is clear; what we need to do is analyze the running time. The runtime is

$$(\#\text{times we run NT}) \cdot (\text{time of NT on a } g\text{-size graph})$$

(really the graph size is $3g$, but it doesn't matter). We assumed that the latter term is $g^{3-\varepsilon}$ (for some ε). And for the first term (the number of times we run NT), there's two types of times when we run NT. One is when we *find* a negative triangle, and one is when the triple no longer has a negative triangle. So it's

$$\#\text{triples} + \#\text{negative triangles found}$$

(the first term corresponds to the case where no negative triangle is found) — every time we run the NT algorithm, it either spits out a negative triangle, or we determine some triple doesn't have some negative triangle (and we move on to the next triple).

The number of triples is at most $(n/g)^3$. And the number of negative triangles found is at most n^2 — because every time w efind a negative triangle, we set an entry D_{xy} to 1 and remove that edge. So every entry of the output matrix is set at most once, and the total number of NTs found is at most n^2.

Now our runtime is

$$\left(\frac{n}{g}\right)^3 + n^2 + g^{3-\varepsilon}.$$

To optimize this, we should set $g = n^{1/3}$; then the runtime is

$$O(n^2 \cdot n^{1-\varepsilon/3}) = O(n^{3-\varepsilon/3}).$$

So there's a bit of overhead, but ay improvement over n^3 gives you one for all-pairs.

The point is basically that this is a decision problem, so as soon as we find *one* witness for an edge, we don't care about that edge anymore and we can discard it. Whenever you can do that, you have a reduction like this.

> **Remark 8.13.** In the notes, we show that NT actually can be reduced to many other problems — for example, if you want to compute the radius of a grpah or the median of a graph, you can take NT and reduce it to them. So to solve APSP, all you need to do is get an algorithm to compute the radius of the graph, or some other property which is just a single integer.

So what this shows you is that when the weights are on the edges, the problem becomes quite hard — we don't know how to solve it fast. What people end up doing is we end up relaxing the problem — we get approximation algorithms and so on. In the next few lectures we'll see how you can get approximation algorithms that are faster, and often don't even use matrix multiplication.

§9 March 6, 2025

§10 March 11, 2025 — girth approximation

Today we'll continue with shortest cycle approximation. As a reminder, this is an interesting graph problem, but we also know its exact computation is essentially equivalent to BMM, so to get fast algorithms we need

to avoid that. So we'd like to consider approximations. Last time we considered additive approximations — where your error is c, and we want an approximation g' satisfying $g \leq g' \leq g + c$. Today we'll consider multiplicative approximations — where you have some $\alpha \geq 1$, and we want some estimate g' with $g \leq g' \leq \alpha g'$.

Definition 10.1. The girth is the length of the shortest cycle in the graph.

Last time, we showed the following theorem.

Theorem 10.2
Girth has a $+1$-approximation in $O(n^2)$ time.

Today we're going to show two results.

Theorem 10.3
Girth has a $+3$-approximation in $\widetilde{O}(n^3/m)$ time.

This is a weird runtime — the denser the graph gets, the faster the algorithm. In the sparse regime (where $m \approx n$, which is the most interesting case), it'll be n^2.

The other result is a multiplicative approximation.

Theorem 10.4
Girth has a multiplicative 2-approximation in $\widetilde{O}(n^{5/3})$ time.

Interestingly $5/3$ is better than n^2, but now we have a multiplicative 2 instead of additive 1.

§10.1 Tools

We'll start with the tools. One tool is what we developed last time — the simple BFS-Cycle algorithm, which we'll recap. The other is a statement from combinatorics about the existence of cycles in dense enough graphs.

§10.1.1 BFS-Cycle

The first tool is the algorithm BFS-Cycle(s) (where s is the starting vertex), which essentially runs BFS until you visit a node twice. Then you can backtrack (as long as you keep predecessor pointers in your BFS tree) and get a cycle out of it.

Last time, we said:

Claim 10.5 — If s is on a q-cycle, then BFS-Cycle(s) will return a cycle of length at most $q + 1$. Furthermore, if q is even, it'll return a cycle of length at most q.

And of course, this thing runs in $O(n)$ time; so it's very efficient.

Proof. You run BFS from s and build up some layers; and at some point you reach the layer $\lceil q/2 \rceil - 1$. And either you've already finished before this layer (so you found a cycle that was very short), or you will definitely finish by exploring the edges out of vertices here. You'll either finish by visiting the same vertex

twice in the next layer, or in this layer; either way you'll close some cycle by going up the tree, and the length of the cycle will either be $2(\lceil q/2 \rceil - 1) + 1$ or $2(\lceil q/2 \rceil - 1) + 2$. Either way, this is at most $2\lceil q/2 \rceil$; this is at most $q + 1$, and if q is even it's at most q. □

We'll change this claim to make it a bit more general. Suppose I run BFS-Cycle but s isn't on a cycle at all. But suppose it's *close* to a cycle — so there's some cycle C, and s is a distance ℓ from C.

Claim 10.6 — If s is at distance at most ℓ from a q-cycle, then BFS-Cycle(s) returns a cycle of length at most $2\ell + q + 1$, or at most $q + 2\ell$ if q is even.

This is roughly true because if you think of this (s to the cycle plus C) as a degenerate cycle, it looks kind of like a cycle of length $2\ell + q$ (ℓ to go to the cycle, q around the cycle, and ℓ to come back). So using the same argument, when the BFS runs, we can say either we've reached a cycle before $\ell + \lceil q/2 \rceil - 1$ (in which case we're done), or we'll definitely close it once we reach the next level — because that degenerate cycle is in our BFS tree. If we go up to layer ℓ, and at some point we have our cycle; this degenerate cycle still exists here, so in the BFS it'll be an option to close. So the same argument works, and we can prove this claim. So even if s is not on a cycle, as long as it's close to a cycle, you still get some cycle by running BFS-Cycle from s.

This turns out to be very useful, as we'll see in a little bit. So this is our first tool.

§10.1.2 Cycles in graphs with many edges

Our second tool is a theorem we mentioned last time:

Theorem 10.7 (Bondy–Simonovits)
For any integer $k \geq 2$, if an n-node graph G has at least $100kn^{1+1/k}$ edges, then G contains a $2k$-cycle.

So as long as there's enough edges (some constant times $n^{1+1/k}$), you can guarantee G contains a $2k$-cycle. In fact, it has many $2k$-cycles. (The constant 100 has been improved over the years; right now, maybe it's 3. But we'll just use 100.)

We're not going to prove this, but we'll see a corollary.

Claim 10.8 — If an n-node graph G has at least $200kn^{1+1/k}$ edges, then a random edge of G is on a $2k$-cycle with probability at least $\frac{1}{2}$.

The idea is you can try deleting edges that are on $2k$ cycles. And you have to delete a lot of edges before you get rid of all $2k$-cycles — you have to delete at least half of them to get below $100kn^{1+1/k}$. And all those edges are on $2k$-cycles.

Proof. To rephrase this a bit, take a maximal set of edges so that when you look at the subgraph induced by these edges, there's no induced cycles. So letting $G = (V, E)$, let $E' \subseteq E$ be maximal such that the subgraph induced by E' has no $2k$-cycle.

So we have G, and somewhere inside G, we have this subset E'. We know E' is inclusion-maximal, so if I add any other edge e, then all of a sudden $E' \cup \{e\}$ has a $2k$-cycle. This means every edge not in E' must create a $2k$-cycle. So every $e \in E \setminus E'$ is in a $2k$-cycle (because E' was maximal).

Now, we must have $|E'| \leq 100kn^{1+1/k} \leq \frac{1}{2}|E|$ by the Bondy–Simonovits theorem. That means $|E \setminus E'| \geq \frac{1}{2}|E|$. So if I pick a random edge e, with probability at least $\frac{1}{2}$ it ends up in $E \setminus E'$, and thus is in a $2k$-cycle. □

Given this claim, suppose I give you a graph that has a lot of edges (at least $200kn^{1+1/k}$). How can I find a $2k$-cycle in this graph very fast (with constant probability)? How do I find a $2k$-cycle in linear time in n with probability $\frac{1}{2}$? We select a random edge and run BFS-Cycle from both of its endpoints. We know this edge will be in a $2k$-cycle, and the BFS-Cycle will find a cycle of length at most $2k$ (because $2k$ is even).

So this is another simple tool.

Claim 10.9 — If G has at least $200kn^{1+1/k}$ edges, then in $O(n)$ time, we can find a cycle of length at most $2k$ with probability $\frac{1}{2}$.

Proof. As we said, pick a random edge e, and run BFS-Cycle from both its endpoints. One of these will find a cycle of length at most $2k$. (You could even just do it from one endpoint; it doesn't matter.) □

You can also boost this — you keep doing it for $O(\log n)$ steps, and in $O(n \log n)$ time, you can find a cycle of length at most $2k$ with probability $1 - 1/\mathsf{poly}(n)$.

Remark 10.10. This is a very simple tool, but we don't know how to derandomize it (without paying much more on the time).

§10.2 An additive 3-approximation

Now we have both tools we need; and we'll get to the $\widetilde{O}(n^3/m)$ +3-approximation algorithm.

This algorithm is kind of — you have a few cases. There's the case where m is very small, in which case you just run the +1-approximation algorithm and it already works. If k is not too small, then you can kind of guarantee there's some density where the Bondy–Simonovits theorem kicks in, and either the +1-approximation already works or you have an upper bound on your girth; and if you have that, you can play with it (now you can do low-degree high-degree tricks). This is roughly a summary. So let's start with the simple cases.

Let $k' = \lceil (\log n)/(\log\log n) \rceil$. Notably, $n^{1/k'} \leq n^{(\log\log n)/(\log n)} = 2^{(\log n \log\log n)/\log n} = \log n$.

Then we have some cases.

Case 1 ($m < 200k'n^{1+1/k'} \leq O(n \log n)$). In this case, m is very small. So now we can run the +1-approximation. This takes $O(n^2)$ time; but $n \lesssim m/(\log n)$, so this is at most $\widetilde{O}(n^3/m)$. So in this case, we're already good.

Case 2 (There exists an integer k such that $200(k+1)n^{1+1/(k+1)} \leq m < 200kn^{1+1/k}$). In our proof, we'll just talk about k.

Previously (in Claim 10.9), we saw that in $\widetilde{O}(n \log n)$ time, we can find a $(2k+2)$-cycle with high probability — since this says $m \geq 200(k+1)n^{1+1/(k+1)}$, so I can pick a random edge and run my BFS-Cycle algorithm $O(\log n)$ times, and return the shortest cycle I've found; so with high probability, in $O(n \log n)$ time I'll with high probability return at most a $(2k+2)$-cycle.

This is very fast, and gives a good approximation if the girth is big enough — if $g \geq 2k-1$, then this is already a +3-approximation. So if the shortest cycle returned by running this randomized procedure is *not* a +3-approximation, then the girth must be small.

So if we're not done, then we must have $g \leq 2k-2$.

Now let C be the shortest cycle. So we draw C. There's two cases — either C has a vertex which has high degree, or it doesn't.

First suppose C has a vertex of high degree:

Case 2a (C has a node of degree at least $n^{1/k}$).Then we claim if we had a hitting set of the neighborhoods of all vertices of high degree, then we can find a node s which is *close* to the shortest cycle.

So we let S be a hitting set of $N(v)$ for every v with $\deg(v) \geq n^{1/k}$, with $|S| = \widetilde{O}(n^{1-1/k})$. Then S contains a node s at distance at most 1 from C.

(We're hitting sets of size $n^{1/k}$, so we need a hitting set of $n/n^{1/k} \cdot \log n$.)

Now we can use Claim 10.6 about BFS-Cycle — so we run BFS-Cycle(s) for every $s \in S$, and we'll get a cycle of length at most $2 \cdot 1 + g + 1 = g + 3$ (the $+1$ is because if the girth is odd, you might make a mistake — that's the $+1$ in Claim 10.6).

And the runtime here is that for every node in S, you're running BFS-Cycle; so the runtime is roughly $n^{2-1/k}$. But we know $m \leq 200kn^{1+1/k}$; this means this runtime is $\widetilde{O}(n^3/m)$.

So now what's left? We've handled the case where C has a vertex of high degree; now we need to deal with the case where all the vertices in the cycle have low degree. This is actually very simple to do.

Case 2b (C only has nodes of degree at most $n^{1/k}$, and $g \leq 2k-2$). As a hint, suppose your cycle has length *exactly* $2k-2$. Then it looks like two paths which are disjoint and have length $k-1$. (If it's less, then the two paths might not be disjoint.)

So here's what we can do: we remove all vertices of degree bigger than $n^{1/k}$ (the shortest cycle doesn't need them). Then for every vertex, we enumerate all the paths of length $k-1$ from it. How many paths of $k-1$ are there from a particular vertex? At most $n^{(k-1)/k}$.

So we fix some v; then the number of $(k-1)$-length paths starting from v is at most $(n^{1/k})^{k-1} = n^{1-1/k}$ (because I start at v; there's $n^{1/k}$ choices for the next vertex; from each there's $n^{1/k}$ choices for the next; and we only go up to $k-1$ levels).

So I take each vertex and enumerate these paths. Then all I need to do is look at pairs of paths that are not the same, and I'll find the shortest cycle. So the way I do it is I take these paths; I know that if I'm looking for a shortest cycle, they have to start at the same vertex. So for each vertex v, I sort them by the ending vertex, and then I check whether they're disjoint. If they're disjoint, I find a cycle. And I can also do it for every shorter path length (there's only k^2 choices for the shorter lengths, and k is a constant).

So we can sort by endpoints to check for two disjoint paths starting and ending at the same vertex. You can do this for $(k-1)$-length paths, but you can also try k'-paths and k''-paths for all $k', k'' \leq k-1$; the number of choices for these is k^2, which is a constant. So you try them all, do the sorting, and take the shortest cycle you can find this way.

So the total runtime is $n \cdot n^{1-1/k} = \widetilde{O}(n^{2-1/k})$ (the tilde is for the sorting).

In this case we solved the problem exactly — we don't get any error (this is an exact computation of the shortest cycle).

The runtime is again $\widetilde{O}(n^{2-1/k})$, and $m \leq 200kn^{1+1/k}$, so this is at most $\widetilde{O}(n^3/m)$.

So in all cases, our runtime is $\widetilde{O}(n^3/m)$.

> **Student Question.** *Why does there always exist k like this?*
>
> **Answer.** We know k' exists — there's some k' where $m \geq 200k'n^{1+1/k'}$. Then you decrease k' by 1 and try again. At some point, because we know $m \leq n^2$, $k = 1$ will give you an upper bound. So you start with k' and decrease it until you get to 1; and at 1, you know the upper bound is satisfied.

> **Student Question.** *How do you know which k to use, if we don't have time to count all the edges?*
>
> **Answer.** We're already using randomness. So we can kind of estimate what m is by sampling — you can estimate m within some constant factor, and you don't need too much time to do that.

§10.3 A multiplicative 2-approximation

The next thing is to use basically the same tools, but to give a multiplicative approximation that's faster than n^2 time. (This algorithm will run in at least n^2 time, because we're running an additive $+1$-approximation in one of the cases. But with a multiplicative approximation, we can avoid this.)

Remark 10.11. In fact, $n^{5/3}$ is the best-known runtime for a multiplicative 2-approximation (if you could do better, that would be amazing).

Suppose that $g = 4c - z$ for $c \geq 1$ and $z \in \{0, 1, 2, 3\}$ — basically, we think about g mod 4, and we put a $-z$ instead of a $+$. (If c happens to be 1, then z has to be 0 or 1, because the girth is always at least 3.)

The reason we wrote it like this is because we'll use something about division by 4. We're going to prove the following.

Theorem 10.12

If the girth of G is $g = 4c - z$, then in $\widetilde{O}(n^{5/3})$ time, we can return a cycle of length at most $6c - z$ if g is even, and at most $6c - z + 1$ if g is odd.

Corollary 10.13

We can get a 2-approximation in $\widetilde{O}(n^{5/3})$ time.

This is because if you look at $6c - z$, we can write

$$6c - z + 1 \leq (6c - z) + (2c - z) = 2(4c - z)$$

(because $2c - z \geq 1$ — if $c \geq 2$, then because $z \leq 3$ this is true; and if $c = 1$ then $z \leq 1$, so this is still true).

So as long as I can return a cycle of length at most $6c - z + 1$, I'm returning a cycle of length at most $2g$. In fact, this is usually much better — it's roughly a $\frac{3}{2}$-approximation, but a bit worse because of the $-z$'s. But it's definitely a 2-approximation.

So my goal is to return a $(6c - z)$-cycle (or a $(6c - z + 1)$-cycle if the girth is odd) whenever the girth is $4c - z$.

We're going to define the following. I don't know what c is, but for now let's pretend that we do. Define

$$N^c(v) = \{x \in V \mid d(v, x) \leq c\}.$$

(This is kind of the c-neighborhood of v.)

And then we consider our shortest cycle C. And suppose it contains some vertex v whose c-neighborhood is large. So we look at its c-neighborhood; and suppose this neighborhood is quite large.

Then we can randomly sample using hitting sets, and find a node at distance at most c from the cycle. Then I'm very close to the cycle, so I can immediately return a cycle whose length is at most $4c - z + 2c + 1$ (using BFS-Cycle and Claim 10.6). So this is the first case — where some vertex has very large c-neighborhood.

The second case is when every vertex on your cycle has very small c-neighborhood; and we'll handle that in a different way.

Case 1 (There exists $v \in C$ with $|N^c(v)| \geq R$). (Here R is some parameter.) You can either take a random sample and argue your random sample will hit this thing; but you can also do it deterministically, and we'll do that.

So for every $x \in V$, you run BFS until we visit R vertices. So we have our x, and you find some BFS tree until you have at least R vertices in there. We call this partial BFS tree T_x; and it has size R.

How long does it take us to compute this partial BFS tree (for a particular vertex)? It'll run in $O(R^2)$ time. And we do this for every vertex, so the runtime is $O(nR^2)$. (We'll set R to be small, so this is less than $n^{5/3}$.)

Student Question. *Why does the BFS take $O(R^2)$ instead of $O(R)$?*

Answer. There could be R^2 edges. There are some ways to avoid this, but for vanilla BFS, if you run until you have R vertices, you could have R^2 edges (e.g., if you have a clique). But actually this is not the bottleneck — you can do something to avoid this R^2 — but a different part of the algorithm has an $O(nR^2)$ bottleneck, so we might as well do this.

So now I have this partial BFS tree. And we're in the case where there's some $v \in C$ with large c-neighborhood (i.e., $|N^c(v)| \leq R$).

Then we claim that $T_v \subseteq N^c(v)$. This is because T_v is the closest R vertices to v, and we claimed $|N^c(v)| \geq R$; so every vertex in T_v has distance at most c from v.

That's great. So now what we'll do is we take all the vertices in the graph, we take their partial trees we computed, and we get a greedy hittig set of those partial BFS trees. So we let S be a (greedy) hitting set of $\{T_x\}_{x \in V}$; and because $|T_x| = R$, we have $|S| \sim O(\frac{n}{R} \log n)$.

Now we run BFS-Cycle(s) from every $s \in S$. At some point, there'll be some node s that our hitting set S hits, in $N^c(v)$ for our vertex $v \in C$. And therefore you'll be running BFS-Cycle from something at distance at most c from the shortest cycle. So we get a cycle of length at most

$$(4c - z) + 2c + 1 = 6c - z + 1$$

(where $4c - z$ is the length of C, and $2c + 1$ is the error of BFS-Cycle). And if g is even, you don't suffer the $+1$.

We also have to say what the runtime is; we have $|S| = \widetilde{O}(n/R)$, so this is $\widetilde{O}(n^2/R)$.

Now we only have one case remaining — that for every vertex on the shortest cycle, its c-neighborhood is actually small.

Case 2 (For all $v \in C$, we have $|N^c(v)| \leq R$). This means for every $v \in C$, we have $N^c(v) \subseteq T_v$ (it's the opposite of what we had before). In particular, for every vertex in the c-neighborhood of a vertex on the cycle, we've computed its distance to everything in the c-neighborhood.

Now we'll justify why we can use c here. We said the girth is $4c - z$, so there's 4 choices for this — $4c$, $4c-1$, $4c-2$, and $4c-3$. In each of these cases, I can find four vertices a, b, x, and y (in the order a, x, b, y on the cycle); in the $4c$ case the distances will all be c, in the $4c-1$ case they'll be c, c, c, $c-1$; and so on.

So there'll be four vertices where x and y are in the partial BFS tree under a, and also the partial BFS tree under b (and I've computed the distances from a to x and y, and similarly from b to x and y).

So we can find 4 nodes on C, roughly equally spaced, such that $x, y \in T_a$ and $x, y \in T_b$. This means we've computed the distances between x and y through a, and also through b.

We've computed all these distances, and we kind of have to find the best way to combine these paths through trees to get the shortest cycle.

So let's define the following.

Notation 10.14. For a node v, we define $d_v(x, y)$ as the distance in T_v between x and y.

So we have some vertex v and its partial BFS tree, and x and y are somewhere in it; and we consider the distance in the tree.

What I want to do is look at all the BFS trees I computed, look at all the pairs (x, y), and find the minimum sum of two distances, one in one tree and one in the other; and I want to somehow claim that minimum sum will give me the shortest cycle. That's roughly the idea.

We'll give a first attempt, and say why it doesn't work; and then we'll fix it.

Here's what we're going to do: We'll first build some dictionary (e.g., a hash table or a binary search tree or whatever — we're okay with paying $\log n$ to search). So we build some dictionary Q, which will eventually be indexed by pairs of vertices (x, y). How do we build it? For every vertex $v \in V$, you look at T_v (its partial BFS tree); and for every pair of vertices $x, y \in T_v$, you insert $(d_v(x, y), v)$ into $Q[x, y]$. (By 'insert' we mean you tack it on — so each $Q[x, y]$ is a list, and we append this to the end of the list.)

This means I look at each pair in T_v, find its distance in the tree (which I can compute very easily), and I stick in v together with the distance in position $[x, y]$.

How long does this take? It kind of depends on how many triples (v, x, y) there are; and we already argued the number of those triples is nR^2, because $|T_v| \leq R$ (so the number of pairs (x, y) is R^2, and the number of v's is n). So this takes $\widetilde{O}(nR^2)$ time.

After we do this, for each $[x, y]$, I'll sort the list I built up by the distance. So for each $[x, y]$ in Q, we sort $Q[x, y]$ by the first piece $d_v(x, y)$.

Now you see if there's some v with some distance, and v' with some other distance — Attempt 1 is to take the first two elements in the sorted list $Q[x, y]$, and claim this forms a shortest cycle (and then you take a min over (x, y)).

So you look at each $Q[x, y]$ and take the smallest two elements; and then you minimize over $[x, y]$, and you claim you've found the shortest cycle in the graph.

Does this work? Well, one issue is, suppose the graph is just a path from x to y. And you pick one v somewhere in the middle of the path, and another v' somewhere else on the path. If you put these paths together, you're not getting a cycle.

So this fails. However, we can look at the following thing. In my cycle (which we know exists for some x and y), if in my tree I keep the next node after x on the path from x to y, then the two nodes will be different for the two cycles. So I can change this very mildly so that it all of a sudden will work — in addition to keeping $d_v(x, y)$, I also keep the successor of x on this shortest tree-path from x to y.

So in T_v, for every (x, y), we also look up $p_v(x, y)$, which we define as the successor of x on the tree path from x to y. So in your tree, it could be that x is somewhere and y is somewhere else; then $p_v(x, y)$ is the node above x. Or it could be that y is a descendant of x, and then p is the node under x (on that path).

So we also keep track of this. And now we'll change the algorithm a little bit — we insert $(d_v(x, y), p_v(x, y), v)$ into $Q[x, y]$. And we still sort by $d_v(x, y)$; but instead of just taking the first two elements, we take the first two elements with *different* $p_v(x, y)$'s.

So this means we have some list $Q[x, y]$; and there's some (d_1, v_1, p_1), $\ldots$, (d_k, v_k, p_1), $\ldots$, and finally at some point we get (d', v', p_2) where $p_2 \neq p_1$. So you take the first one of these (the first one with a different p).

And now we claim that this will work. Why? I've found some v with x and y in its tree, and I've found some v'. And there's two paths from x to y, one through the white tree and one through the yellow tree, and they differ on the first node after x. If they differ on the first node, then they actually close a cycle somewhere. They might be completely disjoint, or you can find the first place where they meet, and you get a cycle. And the length of that cycle will be at most the sum of the two path lengths.

So no matter which two elements minimize the sum, I'll get some cycle of at most that length — if (d', v', p') and (d'', v'', p'') minimize $d' + d''$ over all x and y, then we get a cycle of length at most $d' + d''$.

And in our case, for the particular choices of a, b, x, and y on the shortest cycle (where C is the shortest cycle and we drew a, b, x, and y), the $Q[x, y]$ list is going to contain (d_a, a, x') and (d_b, b, x''), where $x' \neq x''$. And we'll have $d_a + d_b = g$ (this is the length of the shortest cycle). So this is a valid choice; and therefore the two things that we return — $d' + d''$ — will be at most g. And because you always have a cycle of at most that length, you'll actually find the exact shortest cycle here.

All this is completely deterministic (you can use a deterministic dictionary if we don't care about log factors), and the runtime here is $\widetilde{O}(nR^2)$ (the number of triples — a vertex and two things in its BFS tree).

So the final runtime is $nR^2 + n^2/R$ (where n^2/R is where one of the vertices on the shortest cycle had a large c-neighborhood; and nR^2 is to compute the partial BFS trees and handle the case where every vertex has a small c-neighborhood). So you set $R = n^{1/3}$ and get your runtime of $n^{5/3}$.

> **Question 10.15** (Open). Improve the runtime for a 2-approximation.

> **Question 10.16** (Open). Is there a $(2 - \varepsilon)$-approximation in $n^{2-\delta}$-time?

Really the main difficulty there is the special case where the girth is 3. If the girth is 3, how do you find a 3, 4, or 5 cycle in faster than quadratic time? We don't know.

§11 March 13, 2025 — Graph diameter

Today we're going to talk about one of Virginia's favorite problems, computing the diameter of a graph — the maximum shortest-path distance.

As usual, we write $G = (V, E)$, $n = |V|$, and $m = |E|$. We'll have a weight function $w : E \to \{1, \ldots, n^c\}$ (for some constant c).

> **Definition 11.1.** The *diameter* of G is $\max_{u,v \in V} d(u, v)$.

This is a very nice measure of how information spreads through networks. For example, in distributed algorithms, to send a piece of information through the whole network, you need to spend at least this many rounds.

§11.1 Diameter vs. APSP

Given the definition, there's a simple way to compute the diameter — just compute all the distances and take the maximum. This requires computing APSP, which is a very expensive thing to compute. Unfortunately, when you want to compute the diameter *exactly*, this is the only algorithm we have; so the runtime is $n^3/\exp(\sqrt{\log n})$ in terms of n, and if your graph is sparse you can do a bit better ($mn + n^2 \log\log n$).

Now, there's a few questions.

> **Question 11.2.** Why do we need to solve APSP? In the dense case, can we hope to do better?

> **Question 11.3.** If we can't, can we show that you can reduce APSP to diameter — so that any algorithm solving diameter faster than cubic time would also solve APSP?

This would be similar to what we showed for Negative Triangle — there we showed if NT had a subcubic algorithm, so would APSP. But such a reduction is difficult to obtain. Today we'll discuss why — we'll give some vague barrier to getting such a reduction.

For the sparse case where we have runtime $mn + n^2 \log\log n$ for APSP, you have to output n^2 distances, so the APSP output has size n^2. But the diameter output has size 1. When m is close to n, you get n^2 runtime (with some polylog), which would be optimal for APSP because you have to output n^2 distances. But it's completely unclear why you would need n^2 time for diameter, if you're only computing one distance.

But decision problems can be equivalent to function problems — negative triangle has a single-bit output, but we saw it's equivalent to APSP. So it's plausible this is as hard as APSP.

But what's interesting is this argument about the output size holds even for approximations. If you want to compute *approximate* shortest path distances, you still need n^2 runtime, because you have to output them. But for *approximate diameter*, maybe then you could get a faster algorithm in the sparse case.

So here's our two goals for today.

Goal 11.4. For dense graphs, give evidence that diameter might be easier than APSP.

Goal 11.5. For sparse graphs, get algorithms for diameter approximation that run in faster than quadratic time — i.e., get $O(n^{2-\varepsilon})$ time approximation algorithms for diameter.

This would again show that diameter, in the approximation setting, is easier than APSP — because for APSP you still have to output n^2 things, so n^2 is certainly the best you can do.

§11.2 Roadmap for Goal 11.4

We're going to start wth Goal 11.4. It's going to potentially be a bit weird, because we'll be talking about nondeterministic algorithms. Back in the day when you looked at P vs. NP, in the definition of NP there was something called a nondeterministic verifier. We'll use that here, but instead of talking about hard problems, we'll talk about problems with polynomial-time algorithms; and we'll talk about the runtimes of their nondeterministic algorithms.

So we're going to start talking about what a nondeterministic algorithm is for a decision problem. Then we'll define some decision problems that capture Diameter and APSP; and we'll argue that these two problems look very different when it comes to nondeterminism. And we'll then try to connect this to why we think there might not be a reduction from APSP to Diameter, and Diameter might actually be easier.

§11.3 Nondeterministic algorithms

Definition 11.6. Let $\mathcal{Q}$ be a decision problem. A *nondeterministic algorithm* for $\mathcal{Q}$ is a deterministic 'verifier' algorithm $\mathcal{V}$ with input (x, π), where x is the input to $\mathcal{Q}$ and π is called a 'proof', such that:

- If x is a YES instance of $\mathcal{Q}$, then there exists π such that $\mathcal{V}(x, \pi)$ outputs YES.
- If x is a NO instance, then for every π, $\mathcal{V}(x, \pi)$ outputs NO.

For example, if $\mathcal{Q}$ is the question 'does the graph contain a triangle,' then an input x to $\mathcal{Q}$ would be a graph. We think of π as a proof — it's a string of bits, which could represent anything.

If $\mathcal{Q}$ is 'does the graph contain a triangle,' then a YES instance is a graph containing a triangle.

We measure runtime just in terms of the size of x (which we call n). This means in particular the length of this proof π has to be a function of n. So if $|x| = n$, we say the runtime of $\mathcal{V}$ is $t(n)$; in particular, this means we should have $|\pi| \leq t(n)$ (for all π the verifier ever considers).

Example 11.7

NP ('nondeterministic polynomial time') is the set of problems which have verifiers running in polynomial time.

So there should be some proof of polynomial length such that the verifier can check it in polynomial time and verify that the answer is yes.

§11.4 Nondeterministic algorithms for Diameter

Diameter is not a decision problem, so we're going to define two decision problems.

Problem 11.8

Given input (G, D), is $\mathsf{Diam}(G) > D$?

Problem 11.9

Given input (G, D), is $\mathsf{Diam}(G) < D$?

The first case means, does there exist $u, v \in V$ such that $d(u, v) > D$? And the second means, is it true that for all $u, v \in V$, we have $d(u, v) \leq D$?

These two decision problems actually have pretty efficient nondeterministic algorithms.

First, what's a nondeterministic algorithm for 11.8? Here π will be u. Then if you give the verifier $\mathcal{V}(G, \{u\})$, it'll run $\mathsf{SSSP}(u)$ and check that there's some v whose distance from u is bigger than D — so it'll return YES if and only if there exists v with $d(u, v) > D$.

If there is a pair of vertices with distance bigger than D, then there exists some u (namely that one) such that when you run this algorithm, you'll find v with distance to u bigger than v. So this is a nondeterministic algorithm.

What is its runtime? It's Djikstra's (we're focusing on the case where the weights are positive), so it's $O(m + n \log n)$. So this is very fast — essentially linear in m.

Now we'll look at Problem 11.9. Here's what we're going to do — imagine that your proof π is going to be long — it's going to have size n^2 instead of 1. But I'm okay with that, because I'm okay with my verification algorithm running in n^2 time.

So π will be: For every $v \in V$, I'm going to report what's supposedly a shortest path tree T_v rooted at v. So there's v at our root, and you have your tree. The size of this tree is linear in the number of vertices (it's just a tree, and it has some weights); so $|\pi| = O(n^2)$.

Now the verifier is going to look at each tree. First, it's going to check that for every $v \in V$, every $x \in T_v$ is a vertex in G, and every edge $e \in T_v$ is an edge in G with the correct weight. Now, that means every path in the tree (from v to some vertex x) is also a path in your graph.

And we're also going to check that for every $x \in V$, we have $x \in T_v$ (so for every vertex in your graph, there is some path in the tree from v to x) and that $d_{T_v}(v, x) \leq D$.

This does *not* check that T_v is actually a shortest path tree; but it checks that T_v contains a real path in the graph from v to x of weight at most D.

And that means for every $u, v \in V$ there exists some path between u and v of weight at most D, so $d(u, v) \leq D$. You don't compute the exact distance, but you check that it's at most D.

So the verifier runtime is actually just $O(n^2)$ — you read the tree for every vertex, check it, and use dynamic programming to compute the distances.

Student Question. *What's the difference between conondeterministic and nondeterministic?*

Answer. Diameter is defined as finding a distance at least D; so we say Problem 11.8 is the nondeterministic version (it's still $\exists\exists$), and Problem 11.9 is the conondeterministic version (where we flipped the quantifiers to $\forall\forall$).

So in either case, we get a very fast nondeterministic algorithm (with some guessing) — much faster than n^3.

§11.5 Nondeterministic algorithms for APSP

Now we're going to try to do something similar with APSP. APSP is very different from a decision problem, but we already know it's equivalent to a decision problem, so we'll work with that instead.

Previously, we saw that APSP is equivalent to the decision problem Negative Triangle:

Problem 11.10 (Negative Triangle)
Given $G = (V, E)$ and weights $w : E \to \{-n^c, \ldots, n^c\}$, is there a triangle $abe \in V$ such that $w(a, b) + w(b, e) + w(a, e) < 0$?

This is a nice decision problem. We're going to try to give a nondeterministic and conondeterministic algorithm for this thing. But it turns out the only way we know how to do this is use a slightly harder problem than Negative Triangle.

Problem 11.11 (Zero Triangle)
Given G and w, is there a triangle abe such that $w(a, b) + w(b, e) + w(a, e) = 0$?

So the $<$ becomes $=$.

It turns out that you can reduce Negative Triangle to Zero Triangle. How do you do it? Well, you already did all the work in one of your problem sets — you reduced the dominance product to an equality product, which is essentially reducing a $\leq$ to an $=$ via some $\log n$ trials (looking at the bit representations of the weights). You can do basically the same thing here — you look at the bit representation of teh weights. There's $O(\log n)$ bits, and for each bit you create an instance of the zero triangle problem. If you solve all these instances, you get a solution to negative triangle.

So Zero Triangle is at least as hard as Negative Triangle; so if we can get some algorithm for it, we can also solve NT and APSP. The known nondeterministic algorithms for NT all use nondeterministic algorithms for ZT (and run them on $\log n$ instances by looking at the bit representations and cutting them off).

Exercise 11.12. We can reduce Negative Triangle to Zero Triangle.

Now what we're goign to do is take this Zero Triangle problem and look at two versions.

The nondeterministic one:

Problem 11.13
Does there exist abe such that $w(\Delta_{abe}) = 0$?

The conondeterministic one:

Problem 11.14

Is it true that for all abe, we have $w(\Delta_{abe}) \neq 0$?

The first has a very simple nondeterministic algorithm: the proof is just $\pi = \{a, b, e\}$, and the verifier just checks that it's a triangle and sums up the weights on the edges. In the Word-RAM model, this is $O(1)$ time.

But the conondeterministic version — verifying that the graph *doesn't* have a triangle of weight 0 — is more complicated. We'll give one, but it'll be an algorithm whose runtime is $O(n^{(3+\omega)/2})$. It'll use matrix multiplication (of course), and it's also more than quadratic. That's very different from the diameter case, where you had something quadratic. And this is the best thing we know for NT (and therefore APSP).

And for some reason, we think matrix multiplication is necessary in this case. So if you had a reduction that reduced APSP to Diameter (which was deterministic and didn't use MM), then you'd get a very nice algorithm for negative triangle as well. This would be very interesting, but we don't know how to do it. So we view this as a kind of barrier for why we don't know how to reduce this to Diameter. (We're talking about nondeterministic rather than regular algorithms, but even checking a graph doesn't have zero triangles seems difficult.)

§11.5.1 Conondeterministic version of Zero Triangle

We'll see this algorithm because it's very cool; it would be great if we didn't need this and there were a n^2 algorithm, but we don't know one.

Problem 11.15

There's an $O(n^{(3+\omega)/2})$ nondeterministic algorithm for the conondeterministic version of Zero Triangle.

So let's do this. We have a graph, and we want to verify every single triangle has a weight that's nonzero. So what we're going to do is we're going to use prime numbers. Imagine we have a prime p, where $p \geq n^{\mu}$ (here μ is some constant).

What we're going to do is we'll count the number of triangles in the graph whose weight is 0 mod p.

Claim 11.16 — We can count the number of triangles in G of weight 0 mod p in $\widetilde{O}(pn^{\omega})$ time.

Why is this interesting? The thing is, if your graph doesn't have triangles of 0 weight, but we can kind of count how many triangles are 0 mod p, these triangles that didn't have 0 weight but now have weight 0 mod p are 'fake zero triangles.' And we'll be able to show that there exists some prime p for which the number of these fake triangles (that weren't zero triangles but are mod p) — we'll say there exists a prime where that number is very small. If there's a prime that makes a very small number of fake triangles, then my proof can be the prime, and the fake triangles. Then we actually count the number of triangles that are zero mod p; and we check the list and see those are all fake triangles and the list has the exact right length. And that verifies there's no *real* triangles.

So let's suppose the claim is true. Then:

(1) We'll show there exists a prime p of size roughly n^{μ} such that the number of fake zero triangles (ones which aren't 0 in G, but are 0 mod p in G) is small, specifically $n^{3-\mu}$.

(2) We'll take $\pi = \{p, L\}$ where L is a list of (at most $n^{3-\mu}$) fake triangles.

(3) The verifier computes the number of triangles which are 0 mod p in G; let's call this number t.

Then it checks that $|L| = t$ and that all $abe \in L$ are distinct and they're fake triangles (i.e., that they're triangles, they have weights different from 0, and their weights are 0 mod p).

So if we know the exact number of triangles which are 0 mod p, and the proof gives us exactly this many fake triangles, this means there's no real zero triangles.

So that's the plan. We have to show there exists some prime that's small; if the prime is small, then we can compute the number of triangles that are 0 mod p quickly. Then we show that the verifier can compute this number and then check the list; and because the list is small (the number of fake triangles is small), this check is also fast.

We'll prove Claim 11.16 first — so we have a prime p and a graph G with weights on its edges, and we want to count the number of triangles which are 0 mod p.

Proof. Here's how we'll do it: For every $(u, v) \in G$, we let $\overline{w}(u, v) = w(u, v) \bmod p$ (so this is some integer in $\{0, \ldots, p-1\}$). Then we define a matrix

$$A_{uv} = x^{\overline{w(u,v)}},$$

where x is some variable (so this is a polynomial of degree at most $p-1$). And then we compute A^3 — A is a matrix whose entries are polynomials. We multiply it by itself in a natural way — whenever we need to multiply two entries, we multiply the corresponding polynomials. The degree increases — originally it was at most $p-1$, and then it becomes at most $3p-3$. So A^3 is a matrix of polynomials of degree $O(p)$.

And multiplying two polynomials of degree d takes $O(d \log d)$ time, via FFT; and adding is $O(d)$ (you can do this pointwise). So you can run your matrix multiplication algorithm, where whenever you need to multiply entries you do polynomial multiplication; then the total runtime of computing A^3 is $\widetilde{O}(pn^\omega)$ (the tilde hides $\log p$ factors).

And what is A^3? Well, A^3_{ii} is going to be some polynomial. And if I look up the coefficient of the polynomial in front of x^0, x^p, and x^{2p}, I claim that these coefficients, when you sum them up, you'll get the number of triangles that go through i and have weight 0 mod p. The coefficient in front of x^0 is the number of triangles going through i whose weight under $\overline{w}$ is 0; and so on. Thisi s because you get

$$\sum_{j,k} x^{\overline{w}(i,j)+\overline{w}(j,k)+\overline{w}(k,i)}.$$

The exponent gives you the weight of the triangle; and when you look at the coefficients of x^0, x^p, and x^{2p}, you'll get the coefficients (i.e., number of triangles) in front of the possible sums that are 0 mod p.

So just by looking them up and summing over all the i's, I'll get exactly the number of triangles whose weight is 0 mod p. □

Now we need to show there's some (small) prime such that the number of fake zero triangles is actually small. So we'll do that next.

Claim 11.17 — There is $p \approx n^\mu$ such that the number of fake 0-triangles is at most $n^{3-\mu}$.

Proof. Consider some fake zero triangle xyz; this means $w(x, y) + w(y, z) + w(x, z)$ is nonzero, but it is 0 mod p.

So now let's just consider some nonzero triangle; and I want to know, for how many primes p can this triangle all of a sudden become fake? So we'll consider how many $p \geq n^\mu$ make xyz a fake zero triangle.

How many primes can there be that are big and make this a fake zero triangle? That thing

$$w(x, y) + w(y, z) + w(x, z)$$

is at most $3n^c$. And because every one of these primes is large, we're asking how many of these primes can possibly divide that, and the answer is at most

$$\frac{\log 3n^c}{\log n^\mu}$$

(you have something that big, take some prime bigger than n^μ and divide it out; then take another; if you have too many of them, their product is more than $3n^c$; so the number of them is at most the number of times you can divide $3n^c$ by n^μ).

And this is a constant (you can take the c and μ out). So there's at most a constant number of primes that can make any *particular* triple become a fake zero triangle.

Now let's consider the interval $[n^\mu, Cn^\mu \log n]$, and take C large enough so that this interval contains at least n^μ primes. This can be done via the prime number theorem, which basically says that if you take $n^\mu \log n$, there's at least n^μ primes that are less than it (so if you take C large enough, you get that this interval contains enough primes).

So now we're going to take these n^μ primes, and we're going to build up a list — we write down p_1, …, p_{n^μ} (these are all our primes). And I'm going to build a list starting with (p_1, Δ_{11}), where Δ_1 is some fake zero triangle mod p_1. And then I put in (p_1, Δ_{12}), and so on. So we have a list where we have some prime p_i and some triangle Δ which is a fake triangle mod p_i in G.

Your particular graph G has at most n^3 triangles, and each has at most a constant number of primes that make it fake. So this list has size $O(n^3)$ — n^3 is the number of triangles in G, and each has $O(1)$ primes making it fake.

Meanwhile, that list contains n^μ distinct primes. So that means there exists a prime p such that the number of triangles Δ such that (p, Δ) is in the list is $O(n^{3-\mu})$, by averaging.

So there's some prime for which the number of fake triangles is at most $O(n^{3-\mu})$. □

So in your proof, you put that prime; then your list of fake triangles is at most $n^{3-\mu}$. We may not be able to compute this prime, but we don't need to for the nondeterministic algorithm — the prime exists, and we give it, and then we list all the triangles for that p. Then the verifier computes the actual number of triangles that are 0 mod that p; and the runtime is $\widetilde{O}(pn^\omega)$ by Claim 11.16. We have $p \leq Cn^\mu \log n$ (because that's the upper end of our interval), so this becomes $\widetilde{O}(n^{\omega+\mu})$. And then the verifier just reads that list and checks that every triple is a triangle with weight nonzero but zero mod p; and therefore L contains all the fake triangles, which means there can't be a real triangle. The time to read the list and check all that is proportional to the length of the list, which is $O(n^{3-\mu})$ — the number of fake triangles.

You set $\mu = (3-\omega)/2$ (as usual), and you get $\widetilde{O}(n^{(3+\omega)/2})$.

> **Student Question.** *So π includes p?*
>
> **Answer.** Yes — π is p and the list of triangles that are nonzero and 0 mod p.

> **Student Question.** *Why do I trust that the proof includes all the triangles?*
>
> **Answer.** Because I computed the total number of triangles which are 0 mod p, and I checked that the list contains this many (and all are nonzero and really are fake triangles). So there can't be any more triangles that are 0 mod p — there can't be anything that's actually 0.

This is the only subcubic algorithm that we have for this problem (and for any version of APSP that we care about).

§11.6 Approximation algorithms for Diameter

So hopefully we've finished Goal 11.4 — we've shown Diameter has very simple nondeterministic algorithms, and the only ones we have for APSP are very complicated. This can be viewed as some sort of vague intuition of why it's so hard to reduce APSP to Diameter. It's not very formal, but it's something.

What we're going to do next is move to Goal 11.5. We're switching gears — it's still Diameter, but we're going to be talking about approximation, and in the sparse case we want $n^{2-\varepsilon}$ time.

Definition 11.18. For $\alpha \geq 1$, a *α-approximation* to D is some $\overline{D}$ such that $D \leq \overline{D} \leq \alpha D$ for minimization problems, and $D/\alpha \leq D' \leq D$ for maximization problems.

We'll also introduce one more notion:

Definition 11.19. An *(α, β)-approximation* (or *mixed* approximation) is $\overline{D}$ such that $D \leq \overline{D} \leq \alpha D + \beta$.

So you can get a multiplicative α-approximation, but with an extra additive error.

The first thing we'll see is a very simple algorithm for Diameter that runs in linear time and gives a 2-approximation. Then we're going to get something slightly more complicated, which will look like this $\alpha D + \beta$ thing. (We might have seen the first before.)

§11.7 A simple 2-approximation for Diameter

We have a graph $G = (V, E)$ and some weights $w : E \to \{1, \ldots, n^c\}$, and we want to compute (approximately) $D = \max_{u,v} d(u, v)$. We don't know D; but here's how we compute a simple 2-approximation.

Algorithm 11.20

Pick an arbitrary vertex v, run Djikstra's algorithm to and from v, and return the maximum distance you find.

We claim this is good enough. Why? Imagine $D = d(a, b)$ for some a and b. We can draw a and b; and we ran Djikstra's into and out of v, so in particular we considered the distance from a into v and v out to b. This means our max distance found is at least $\max(d(a, v), d(v, b))$. And by the triangle inequality, for all a, b, and v we have

$$d(a, b) \leq d(a, v) + d(v, b).$$

In our case, $d(a, b) = D$, and therefore at least one of these is at least $D/2$ — they can't both be less than $D/2$, because if they were then their sum would be less than $D/2$.

So $\max(d(a, v), d(v, b)) \geq D/2$. So when we return the maximum distance here, we'll get that it's at least $D/2$. But it's also at most D, because we're returning a distance, and every distance in the graph is at most D (as long as your approximation algorithm returns a true distance in the graph, it'll always be at most the diameter; so the second part $D' \leq D$ in the definition will always be satisfied if you're returning a true distance — so we can kind of ignore that second part, and all we need to show is we can return a distance that's at least D/α).

And the runtime is just running two Djikstra's, which is $O(m + n \log n)$.

§11.8 A 3/2-approximation

Now what we're going to see is a somewhat more involved algorithm that gives you a 3/2-approximation — instead of something that's at least $D/2$ it'll be $2D/3$. It'll be a bit slower — $m\sqrt{n}$ instead of n. But it's still subquadratic — it's still $n^{2-\varepsilon}$ when the graph is sparse.

We'll return a $(\frac{3}{2}, \frac{3}{2})$-approximation. This means there's some additive error. We're going to actually give an algorithm in unweighted graphs that gives you an actual $\frac{3}{2}$-approximation when the diameter is divisible by 3; from the proof you'll see where the additive error comes in when the diameter isn't divisible by 3 or there are weights.

Theorem 11.21

There is a randomized algorithm running in $\widetilde{O}(m\sqrt{n})$ time in unweighted directed graphs that with high probability returns a $\frac{3}{2}$-approximation to D if D is divisible by 3.

The full theorem, which we're not going to prove, is that you could make this deterministic, you can do it in weighted graphs, and you don't need that D is divisible by 3, but if it's not then you get a $(\frac{3}{2}, \frac{3}{2})$-approximation for unweighted graphs, and $(\frac{3}{2}, O(M))$ for weighted where M is the maximum weight in your graph. So you can get some additive error that depends on your max weight in the weighted case; if it's unweighted and $3 \nmid D$ you get a slight error.

The first thing we're going to do is argue as follows: Let a and b be the diameter endpoints, so $D = d(a, b)$. Look at the ball around a that contains all the vertices at distance at most $D/3$ out of a.

So we look at this ball. And imagine that somehow, I find some node s inside this ball — so I've found some node s with $d(a, s) \leq D/3$. Then I can run BFS (because the graph is unweighted) from s, and I'll consider the distance $d(s, b)$. Because s is very close to a, by the triangle inequality the distance from s to b must be large — if $d(a, s) \leq D/3$, then $d(s, b) \geq 2D/3$, because $d(a, b) \leq d(a, s) + d(s, b)$ (so if this weren't true, we'd get $D < D/3 + 2D/3$).

So as long as somehow I get a hold on some node s that's very close to a and I run BFS from it, I'll get a good approximation.

What's an easy case? Is there an easy case when I can guarantee that I run BFS from some node in that ball?

An easy case is when the ball around a of radius $D/3$ is very large. Imagine that I have at least $\sqrt{n}$ nodes that are close to a. Then how can I guarantee I run BFS from a node close to a? (I don't know what a is, but if the ball is large, what can I do?)

If we've learned anything from this class, it's that if something is large, I can take a random subset and hit it. So if a happens to have a ton of nodes close to it, then I can take a random subset (which is not large) and guarantee I hit it; and then I can run BFS from everything in that subset and get a good approximation.

So for the first step:

(1) Let $S \subseteq V$ be a random subset of size $\Theta(\sqrt{n} \log n)$. Then with high probability, if $\left|N_{D/3}(v)\right| \geq \sqrt{n}$, then $S \cap N_{D/3}(v) \neq \emptyset$. (In particular, if a has a large $D/3$-neighborhood, then we'll hit that neighborhood.)

(2) Run BFS from and to every $s \in S$; this takes $\widetilde{O}(m\sqrt{n})$ time. Then return D', the largest distance found.

If $\left|N_{D/3}(a)\right| < \sqrt{n}$, then we'll have $D' \geq 2D/3$ — so if a has lots of nodes close to it, then D' will already be a good estimate.

Otherwise, if S doesn't hit this $D/3$-neighborhood, then the distance from a to everything in this neighborhood is large, i.e., for every $s \in S$, we have

$$d(a, s) > \frac{D}{3}.$$

(Either S intersects the $D/3$-neighborhood, or everything in S is further than distance $D/3$.)

We're done if our hitting set hits the $D/3$-neighborhood of a. If it doesn't, we'll have to do something else. And this is what we're going to do.

We already ran BFS from all $s \in S$, so we know, for every $v \in V$, the value of

$$d(v, S) = \min_{s \in S} d(v, s).$$

So for every v, we know the closest node in S, and the distance from v to S. In the bad case, the distance from a to S is bigger than $D/3$.

So we let w be the furthest node from S, i.e., the node maximizing $d(v, S)$. This could be a or it could be something else; but we know that because a is very far from the set, so is w. So we know

$$d(w, S) \geq d(a, S) > \frac{D}{3}$$

(in the bad case).

So we have some node w; and we know if we haven't gotten a good approximation already, then w is far from the set.

Now what we're going to do is we'll run BFS from w. And we'll also run BFS from the $\sqrt{n}$ closest nodes from w.

So we have our w, and we ran BFS out of it (in the outwards direction); and that found you the closest $\sqrt{n}$ nodes to w. We'll call this W. And we're also going to run BFS out of those nodes.

Let D'' be the largest distance found when we ran BFS out of w; and let D''' be the largest distance found when we ran BFS out of W. In the algorithm, we'll return $\max(D', D'', D''')$.

Now, I ran BFS to and from $O(\sqrt{n} \log n)$ vertices (from the random subset, w, and the $\sqrt{n}$ closest nodes to W), so the runtime is $O(m\sqrt{n} \log n)$.

But why it works is a completely different story. So let's prove that. (The runtime is clear, but the accuracy is unclear.)

Look at this w. We ran BFS from w, so at some point we looked at b. If $d(w, b) \geq 2D/3$, then the estimate D'' satisfies $D'' \geq 2D/3$ (because when we ran BFS from w, one of the distances we considered was $d(w, b)$, so if b is far from w we're already done).

So assume this fails, meaning that $D'' < 2D/3$; this means $d(w, b) < 2D/3$.

Now look at the $\sqrt{n}$ closest nodes to W.

First of all, we had that S hits the $D/3$-neighborhood of every node with high probability. And therefore, if I look at the shortest path from w to b, it has length less than $2D/3$. And let's look at the node at distance exactly $D/3$ on this shortest path. So this is the shortest path from w to b, and there's a node at distance exactly $D/3$ from it.

So there's osme y at distance exactly $D/3$ from w.

> **Claim 11.22 —** We have $y \in W$.

Proof. Well, we said that S (this random subset) hits the closest $\sqrt{n}$ nodes of every w. But somehow, w is far from the set S. So w is at distance more than $D/3$ from s. That means if I look at its $D/3$-neighborhood, it does not contain a node of S, because it's further from S. That means this neighborhood is small — it can't be of size $\sqrt{n}$. So this size must be less than $\sqrt{n}$, otherwise it would have been hit by S. So the $D/3$-neighborhood is of size less than $\sqrt{n}$, which means this entire neighborhood is contained in W (W was the $\sqrt{n}$ closest nodes, and these are teh closest nodes which are distance at most $D/3$, and they're smaller than $\sqrt{n}$). So this entire thing is contained in W, and y is inside there. □

So now what we get is we've run BFS from every node in W, and in particular from y. But y is at distance exactly $D/3$ from w, and w is distance less than $2D/3$ from b, so that means

$$d(y,b) = d(w,b) - d(w,y) < \frac{2D}{3} - \frac{D}{3} = \frac{D}{3}.$$

And therefore if you run BFS into every node of W, you'll get $d(a,y)$ as one of your estimates. So we computed $d(a,y)$; this means

$$D''' \geq d(a,y) \geq D - \frac{D}{3} = \frac{2D}{3}$$

(where the $D/3$ is the $d(y,b)$).

§12 March 18, 2025

Today, in the first 15 minutes we'll finish the proof from last time — we kind of glossed over a few things because we were going a bit fast. Then we'll move on to combinatorial algorithms for approximating APSP within an additive error of 2 in undirected unweighted graphs, and we'll discuss why that's interesting.

§12.1 Diameter approximation

To remind us of the algorithm we saw last time, we have a directed graph G which is unweighted, and we assume the diameter D is divisible by 3.

- First choose a random sample $S \subseteq V$ of size $\Theta(\sqrt{n}\log n)$.
- Run BFS into and out of each $s \in S$, and let D_1 be the largest distance found.
- For all $v \in V$, we can compute $d(v,S) = \min_{s\in S} d(v,s)$. Let w be the node with the largest $d(w,S)$.
- Run BFS out of w, and let D_2 be the largest distance found.
- Let W be the closest $\sqrt{n}$ nodes out of w.
- For all $x \in W$, run BFS into x; let D_3 be the largest distance found.
- Return $\max\{D_1, D_2, D_3\}$.

We're running BFS out of $\sqrt{n}\log n$ nodes, so the runtime is simple; all we need to do is show this is a good approximation. Anything we return is a distance in the graph, so it's always at *most* the diameter; so we need to show it's always at least $2D/3$.

Claim 12.1 — We always have $\max\{D_1, D_2, D_3\} \geq 2D/3$.

Proof. Assume not. We'll use a and b to denote the two vertices in V such that $d(a,b) = D$ — they're the witnesses of the diameter. And we also have a very important tool, the triangle inequality — for every x, y, z, we have

$$d(x,y) \leq d(x,z) + d(z,x).$$

In particular, if we find some z such that $d(a,z) \leq \varepsilon \cdot D$, then we must have $d(z,b) \geq (1-\varepsilon)\cdot D$. So if we find a node that's close out of a, that node is far into b. And vice versa — if we find a node that's close into b, then it's far out of a. This is the only thing we're going to use.

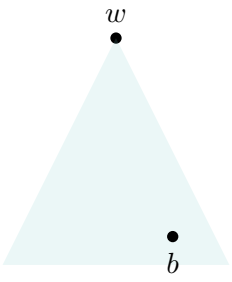
The first thing we're doing is running BFS out of every node in our random sample S. So if there exists $s \in S$ such that $d(a,s) \leq D/3$, then $d(s,b) \geq 2D/3$. Then we'd have $D_1 \geq d(s,b) \geq 2D/3$ (since D_1 includes the distances out of every node in s).

So our assumption $D_1 < 2D/3$ means that for every $a \in S$ we have $d(a, s) > D/3$. This means $d(a, S) > D/3$. And w is the *furthest* node to S, so that also means $d(w, S) \geq d(a, S) > D/3$.

Now we assumed that $D_2 < 2D/3$. This in particular means $d(w, b) < 2D/3$.

Finally, we assumed $D_3 < 2D/3$. We have $w \in W$, so we know that if $d(w, b) \leq D/3$, then by the triangle inequality $d(a, w) \geq 2D/3$, which would mean $D_3 \geq 2D/3$. This is a contradiction, so we must have $d(w, b) > D/3$.

So we have a very good estimate of the distance between w and b — it's between $D/3$ and $2D/3$. Now, if it's bigger than $D/3$ (and $3 \mid D$, and the graph is unweighted), there exists a node y on the shortest path from w to b of distance *exactly* $D/3$.



So we have $d(w, y) = D/3$; notably, this means $d(y, b) = d(w, b) - d(w, y) < 2D/3 - D/3 = D/3$ (because y is on the shortest path).

So we've found a node y that's close to b, which means by the triangle inequality that it must be far from a — we have $d(a, y) > 2D/3$.

Now all that we need to show is that we've actually run BFS into y. Why? We'll show that with high probability we have $y \in W$ — because then we'll have run BFS into y, and D_3 will be good enough.

So let's see why y must be inside W. The only thing we're really going to use about y is that $d(w, y) = D/3$.

The first thing is that with high probability, the set S (which we picked completely at random, with size $\Theta(\sqrt{n} \log n)$) hits the $\sqrt{n}$ closest nodes out of *every* vertex. (It's a big random hitting set.) In particular, there exists $s^* \in S \cap W$ (with high probability). (So W is the $\sqrt{n}$ closest nodes out of w; and S hits every $\sqrt{n}$-neighborhood, so it hits W.)

However, we *also* know that $d(w, S) > D/3$ — we have that w is very far from our set S, so the distance from w to every single node in S is more than $D/3$. In particular, this means $d(w, s^*) > D/3$.

Now let's look at W, which is the $\sqrt{n}$ closest nodes to w. There's some s^*, and we know W contains every node that's closer to w than s^* — so all the nodes at distance less than $d(w, s^*)$ are contained in W (because W contains the closest nodes). But $d(w, s^*) > D/3$; this means every node x with $d(w, x) = D/3$ is in W (because they're closer to w than s^* is). In particular, this means $y \in W$ (it's at distance exactly $D/3$).

So this shows why this algorithm works. □

Remark 12.2. We crucially used the fact that $D/3$ is an integer. Where? We definitely use it when defining y — when we say that because $d(w, b) \geq D/3$, there is a node at distance *exactly* $D/3$. Otherwise you'd have to take floors, and you'd lose an additive error; and if the graph is weighted, you'd lose something based on the mximum edge weight.

But in any case, you can get some mixed approximations (in unweighted graphs with $3 \nmid D$, and in weighted graphs).

Remark 12.3. This is randomized, but you can make it deterministic. What's published has a loss in the runtime, but in unpublished work we have something that completely gets rid of the randomization.

Remark 12.4. For undirected graphs, you can actually get an approximation scheme, where you trade off runtime against approximation quality. We saw a linear time 2-approximation, and for undirected graphs you can get a tradeoff between this and the 2-approximation. It's still open whether you can get an approximation scheme for undirected graphs.

§12.2 APSP in undirected unweighted graphs

Now we'll talk about APSP, and we want to see what we can do without matrix multiplication.

First, if you want to get an additive approximation (an estimate which is at least the distance, and at most the distance plus some additive error):

Claim 12.5 — If we can get a $+1$-approximation for APSP, we can use it to solve BMM.

Proof. Build your graph corresponding to the Boolean matrix product of a matrix A and matrix B; then in BMM we're asking whether the distance between a node on the left and right is exactly 2. But this graph is bipartite, so if the distance isn't 2, then it's at least 4. If you can get approximations which are always at most 1 from the correct answer, then you can distinguish between 2 and 4 for every pair, so you can solve BMM. □

Student Question. *If you get a 3, how can you tell whether the actual value is 2 or 4?*

Answer. An additive error of $+1$ means

$$d(u,v) \leq \widetilde{d}(u,v) \leq d(u,v) + 1.$$

So if it tells you 3, then the actual value has to be 2.

So if we're aiming towards something that doesn't use matrix multiplication, then we have to have an additive error of at least 2. Now we're going to look at additive errors of 2. We'll get two very nice algorithms. The first will be simple; the second will be more complicated, but it's essentially the best-known algorithm without matrix multiplication.

§12.3 Warmup — +2-approximate APSP

As a warmup, we'll see an algorithm for $+2$-approximate APSP in $\widetilde{O}(n^{2.5})$ time. (This is combinatorial, so there's no matrix multiplication.)

We're going to use the triangle inequality, because this is what we know how to use. We're going to look at a shortest path between u and v. We don't know what it is, but for now suppose that shortest path has some vertex x with very high degree.

The only tool we know is how to hit this neighborhood. In fact, we can use a greedy algorithm — we don't need randomness. So we can hit this neighborhood with some set S. And then because this is a length of 1 from our path, we can get a good approximation by estimating the distance from u to s and s to v.

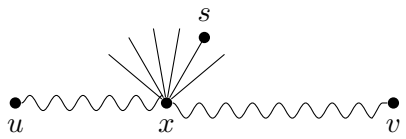
So we say a high-degree vertex is one of degree at least $\sqrt{n}$, and a low-degree vertex is one of degree less than $\sqrt{n}$. We define S to be a hitting set for all $N(x)$ such that x has high degree. You can compute this deterministically in linear time (in the number of edges), with $|S| = \Theta(\sqrt{n}\log n)$.

Now you have your hitting set; and for all $s \in S$, you do BFS at s. Then for all u and v, you set

$$d_1(u,v) = \min_{s \in S}(d(u,s) + d(s,v)).$$

If a shortest path from u to v contains some x of $\deg(x) \geq \sqrt{n}$, then there's some $s \in S$ which is a neighbor of x; this means

$$d(u,v) \leq d(u,s) + d(s,v) \leq d(u,x) + 1 + 1 + d(x,v) = d(u,v) + 2.$$



So this estimate d_1 will be a $+2$-approximation as long as the shortest path between u and v has a vertex of high degree.

What do we do in the other case, to deal with pairs of vertices that don't have high-degree vertices on their shortest paths? We can just delete all the high-degree vertices (or the edges incident to them). Then the remaining graph is sparse — it has $n^{1.5}$ edges. And if you run BFS in the remaining graph (from every vertex), it'll run in $n^{2.5}$ time. This gives some estimate $d_2(u,v)$, which will be correct (i.e., the exact distance) if the shortest $u \rightsquigarrow v$ path has no high degree vertices.

The runtime of this step is $n \cdot n\sqrt{n}$ (the n is for running BFS from every node, and you have $n\sqrt{n}$ edges).

In the first part, you're doing BFS out of $\sqrt{n}\log n$ nodes on the full graph; that takes time $\sqrt{n}\log n \cdot n^2$ (because the graph could be dense).

And in the end, for every pair (u,v), we return

$$\widetilde{d}(u,v) = \min\{d_1(u,v), d_2(u,v)\}.$$

This will always be a $+2$-approximation (and the only time you have an error is when your path has a high-degree vertex — otherwise you're completely correct, because you actually see the whole path).

This is simple and deterministic and you don't have to worry about anything — except for the fact that it's actually not optimal, and we can actually do better.

> **Question 12.6.** Can you do something like this for directed graphs — can you get an additive approximation?

Well, in the BMM construction, if you do the same construction with edges directed left to right, then even distinguishing between *finite* and *infinite* distances solves BMM. And distinguishing between 2 and ∞ can be done with any approximation (multiplicative or additive). So in directed graphs, *any* (additive or multiplicative) approximation can be used to solve BMM. This means directed graphs are hopeless; that's why we focus on undirected graphs. But for undirected graphs, you can get very nice results.

§12.4 A better +2-approximation

Theorem 12.7
There is a $\widetilde{O}(n^{7/3})$ time $+2$-APSP algorithm in undirected unweighted graphs.

This is the best-known algorithm. We'll basically do the same thing from before, but with something slightly different.

When you look at this algorithm, is there something we might want to change? One suggestion is to use Djikstra (instead of BFS). Another is to use recursion. Maybe we can use different hitting set sizes. In our algorithm, we picked only one degree threshold. But maybe we can pick multiple, and have *multiple* hitting sets.

In fact, in the algorithm we'll describe, we'll use Djikstra's and multiple hitting sets. We won't exactly use recursion, but you can use induction (which is basically recursion) to get an approximation *scheme* for this. So we won't use this, but it's very useful.

For $n^{7/3}$, instead of using one degree threshold, we'll use 2. (For the approximation scheme, you use $\log n$ different thresholds.) We'll use thresholds d and D; so *high-degree* means $\deg(v) \geq D$, *low-degree* means $\deg(v) < d$, and *medium-degree* means $\deg(v) \in [d, D)$.

For both of these thresholds, we'll pick hitting sets (for the neighborhoods of vertices with high and medium degrees) — let S be a hitting set for high-degree neighborhoods, so that $|S| \sim \frac{n}{D} \log n$. And let T be a hitting set for medium-degree neighborhoods, so that $|T| \sim \frac{n}{d} \log n$.

Think about $D \sim n^{2/3}$ and $d \sim n^{1/3}$ (but we'll see where these numbers come from). So S is significantly smaller than T — it has size roughly $n^{1/3}$, while T has size roughly $n^{2/3}$.

Because T has size $n^{2/3}$ and we're aiming for an algorithm with $n^{7/3}$ runtime, we can't afford to run BFS from every node in T. We *can*, however, afford to run BFS from every noe in S. So that's the first thing we're going to do: For every $s \in S$, we run BFS, and we define

$$d_1(u, v) = \min_{s \in S}(d(u, s) + d(s, v)).$$

This is a $+2$-approximation if the $u \rightsquigarrow v$ path has a high-degree node, and this runtime is $O(n^2 |S|) = \widetilde{O}(n^3/D)$. So if $D = n^{2/3}$, this is good enough.

Now we can completely forget about the high-degree vertices — we're going to get rid of them. We let G_{med} be G with only the edges incident to low and medium-degree vertices. Then G_{med} has at most nD edges (which, setting $D = n^{2/3}$, is at most $n^{5/3}$).

So it's quite sparse; and we *can* afford to run BFS from every node in T in this sparse graph. So for every $t \in T$, we run BFS from t, but in G_{med}. This computes $d_{\text{med}}(t, v)$ for every $v \in V$.

We *don't* get to compute all the distances between every pair of vertices in medium-degree in G_{med}, because that would be too expensive (this would cost us $n^2 \cdot nD$, which is too much — that'd be $n^{8/3}$). But we *do* get to do it for every $t \in T$. And this runtime is because T has size n/d — so the runtime is

$$\frac{n}{d} \cdot nD \sim n^2 \cdot \frac{D}{d}.$$

So our first runtime was n^3/D, and our second runtime is $n^2 \cdot D/d$. In order to balance these runtimes, we should set $d = \sqrt{D}$; this is why when we set $D = n^{2/3}$, we get $d = n^{1/3}$.

So far, we're not done. The next thing we're going to do is try to combine what we've learned about the medium degree hitting set distsances together with the distances from high degree and so on.

We're going to create, for every vertex $u \in V$, a new graph $G_u = (V, E_u)$ (so it's a graph on the same set of vertices, but with new edges). This graph is going to have both unweighted and weighted edges. And what we want is that

$$d_{G_u}(u, v) \leq d(u, v) + 2 \quad \text{for every } v \in V$$

as long as the $u \rightsquigarrow v$ path has no high-degree vertex.

The first thing we're going to do is define E_u — we know G_u has all the original vertices, but we need to define which edges to include. We want E_u to be very small — in particular we want $|E_u| = O(n \cdot d)$. So what edges from the original graph can I afford to add? I want to add edges from the original graph, and then some extra stuff. If I want only nd edges, for every low-degree vertex I can afford to add its entire neighborhood — so for every low-degree node, we add all its edges.

Besides low-degree vertices, we also have some medium-degree vertices. What do we do with the medium-degree vertices? Well, for each medium-degree vertex x, we know that the hitting set T has a neighbor of x (which is pretty much as good as being x, up to an additive error). So we find our set T. For every medium-degree vertex, I know T contains a neighbor of it; so I pick an arbitrary neighbor of it in T, and add an edge there. So I connect each medium-degree vertex to the hitting set with a single edge. This is very cheap — I add at most n edges.

So for every medium-degree x, I pick an arbitrary neighbor $t \in T$ and add (x, t) to E_u.

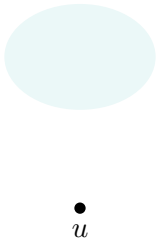
So I've added all the low-degree vertices, and I added a single edge from every medium-degree vertex to the hitting set. That's $nd + n$ edges. So far, I haven't used anything about u — every graph will contain all these edges.

But now I'm going to use u — what can I add to u? I know something about the distances in the graph from u to other vertices, and I want to represent those distances with weighted edges. (I ignore high-degree vertices.)

What distances have I computed that involve u? The distances from u to the vertices in T. I've definitely computed those, so let's add them — from u I add an edge to every single node in T. And I put a weight on this edge, where

$$w(u, t) = d_{\text{med}}(u, t)$$

(this is something I computed already, when running BFS from every $t \in T$ in G_{med}). The number of edges I add here is at most $|T| \leq n$.



(The yellow edges are weighted with d_{med}, and all the other edges have weight 1.)

So this graph is pretty sparse — it has $O(nd)$ edges. And now I'm going to run Djikstra's algorithm from u in G_u, for every u. This gives you $d_u(u, v)$ for every $v \in V$. And then in the end, we're going to return

$$\widetilde{d}(u, v) = \min\{d_1(u, v), d_u(u, v)\}.$$

What's the runtime of this? It's $n \cdot nd \log n$ (where n is the number of vertices, and nd is the number of edges in your graph). This is $n^2 d$ (we don't care about log factors).

The previous runtimes were n^3/D and $n^2 D/d$. And now we have n^3/D. We want to make these three things equal, which means we set $D = n^{2/3}$ and $d = n^{1/3}$. So we do some balancing and get $\widetilde{O}(n^{7/3})$ runtime.

But we still have to prove correctness:

Claim 12.8 — We have $d(u, v) \leq \min\{d_1(u, v), d_u(u, v)\} \leq d(u, v) + 2$.

Proof. The first thing we know is that if the $u \rightsquigarrow v$ node has a high-degree vertex, then we're good — d_1 is already a +2-approximation. So the only thing that remains to show is that if the $u \rightsquigarrow v$ shortest path has no high-degree nodes, then $d_u(u,v) \leq d(u,v) + 2$ (and $d(u,v) \leq d_u(u,v)$).

First, why is $d(u,v) \leq d_u(u,v)$? Any edge we added in G_u has weight at least the corresponding distance in G — it's either an original edge, or it's a weighted edge whose weight is at least the distance between the two endpoints. So the distances here are upper bounds on distances in the original graph.

So what we need to show is that $d_u(u,v) \leq d(u,v) + 2$; and if we can show this, then we'll be done. Now let's prove this.

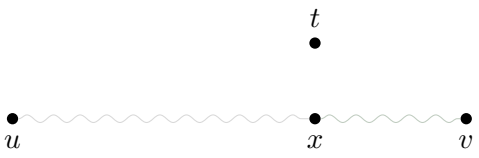
Imagine you have your shortest path between u and v, which we call P_{uv}. It doesn't have high-degree nodes. Suppose it also doesn't have medium-degree nodes. Then what do we know about it? If there's no medium-degree nodes, then it's all low-degree nodes, and in that case we actually get the exact distance — all the low-degree vertices have their entire neighborhoods included, which means the entire path P_{uv} is included in G_u. So when we run Djikstra's, we'll see this path, and we'll get the exact distance. In other words, P_{uv} is contained in G_u, so $d_u(u,v) = d(u,v)$ (we have no error).

So now we can assume P_{uv} does have medium degree nodes. One thing we like to do when there's potentially multiple medium degree nodes is to pick an extremal one. So we'll pick the last one — consider the very last medium-degree node on the shortest path from u to v (the one that's closest to v), and call it x.

It's a medium degree node, so it has a neighbor in T. Let $t \in T$ be a neighbor of x; then we added the edge (x,t) to E_u.

Now, we know that the edge (u,t) is also in E_u, with weight $d_{\text{med}}(u,t)$.

And what do we know about the last edges between x and v? Well, there's no other medium-degree vertices here (because we chose x to be the last one), which means all these edges are included in G_u.



So this yellow edge, red edge, and all the last green edges are in there; this means this path $utx \rightsquigarrow v$ is in G_u.

Now, the portion of the path from u to x is in G_{med}, and so is the edge (x,t). That means

$$d_{\text{med}}(u,t) \leq d(u,x) + 1$$

by the triangle inequality.

So then we get that this $utx \rightsquigarrow v$ path has weight

$$d_{\text{med}}(u,t) + 1 + d(x,v) \leq d(u,x) + 1 + 1 + d(x,v).$$

And $d(u,x) + d(x,v)$ is exactly $d(u,v)$, because x is on the shortest path. And we're done. □

Remark 12.9. The key thing is we made sure we picked the *last* medium-degree node on P_{uv} path so that every edge in the final $x \rightsquigarrow v$ path is in G_u; and (x,t) is also in G_u. And for the edge (u,t), we used the triangle inequality to say that its weight is at most $d(u,x) + 1$. (We used the fact that the graph is undirected because we're using the triangle inequality in both directions.)

And we used Djikstra's algorithm because we needed weights; and we used multiple hitting set sizes.

Remark 12.10. You can keep doing this with multiple degree thresholds. You'll get faster and faster approximations. But the error you get will not be a +2-approximation — you'll accumulate errors. You can get a $+2k$-approximation in some runtime that approaches n^2 (as k grows). (It's always even because you're using the triangle inequality twice.) And in the end, there's an $n^2 \log n$ algorithm that achieves a $+\log n$ approximation. These results are by Dor–Halperin–Zwick, and they're still the best combinatorial algorithms for the additive approximation problem. It's very interesting to improve them — we have been able to improve them using matrix multiplication (though that kind of defeats the point, because we were trying to avoid it).

Remark 12.11. So the message is you can avoid MM with some error. You can also get *multiplicative* approximations for directed graphs using MM. For undirected graphs, you can get multiplicative approximations that are faster — you can get a 2-approximation that's $n^{9/4}$ time or something like that. This is an active area of research, and there's extensions to weighted graphs and so on (as long as there aren't some very simple conditional lower bounds where you can embed matrix multiplication, like we did with directed graphs or the +1-approximation).

§13 March 20, 2025 — Spanners

Today we'll talk about spanners.

§13.1 Spanners

Definition 13.1. A *(α, β)-spanner* of $G = (V, E)$ is a subgraph $H = (V, E_H)$ (with the same vertices and a subset of the edges) such that for all $u, v \in V$ we have

$$d(u, v) \leq d_H(u, v) \leq \alpha d(u, v) + \beta.$$

The first inequality follows just from the fact that H is a subgraph; the interesting part is the upper bound. We want to compute a spanner quickly such that it has much fewer edges than the original graph. So then now that you have the spanner, your distances are roughly preserved; and you can run whatever algorithm you want on the spanner instead of the original graph, which will be faster because the graph is now sparse.

There are special cases:

Definition 13.2. The case where $\beta = 0$ is called *multiplicative spanners* (here we typically just call them *α-spanners*). The case where $\alpha = 1$ is called *additive spanners* (here we call them *$+\beta$-spanners*).

§13.2 Existence statements

Today we're going to prove some *existential* theorems for undirected graphs. We'll then see a bunch of techniques. But what we'll do today won't produce efficient algorithms (we'll give an algorithm, but it might be exponential time).

Here's the two theorems we'll prove:

Theorem 13.3

Every n-node undirected unweighted graph $G = (V, E)$ contains:

- A $+2$ spanner on $\widetilde{O}(n^{1.5})$ edges.
- A $+4$ spanner on $\widetilde{O}(n^{7/5})$ edges.
- A $+6$ spanner on $\widetilde{O}(n^{4/3})$ edges.

Theorem 13.4

For every integer $k \geq 2$, every n-node undirected G with nonnegative edge weights contains a (multiplicative) $(2k-1)$-spanner on $O(n^{1+1/k})$ edges.

So there's two sort of worlds — the additive approximation and multiplicative approximation world. Both are for undirected graphs. Why can't we do anything like this for directed graphs? Directed graphs — what we want to do is prove that there's always some subgraph that has strictly fewer edges and doesn't distort the distances too much. So why are there no nontrivial (α, β)-spanners (with constant — or even finite — α and β) for directed graphs? (Nontrivial means a spanner with fewer edges than the original graph.)

One picture is you could take a complete bipartite graph, and just add all the edges from left to right. Now if I remove any edge, the distance between its endpoints was 1, but it becomes ∞. So omitting any edge (x, y) makes $d_H(x, y) = \infty$, when we originally had $d(x, y) = 1$. So no matter what finite α and β you pick, you can't make $\infty \leq \alpha + \beta$, so there's no way. So there's no nontrivial (α, β)-spanners for directed graphs. Because of this, we'll forget about directed graphs today, and just focus on undirected graphs.

The second question is, when we have an additive spanner, we're focusing on unweighted graphs. Why do we have to do this whenever we want a constant additive error? For any spanner of a weighted graph, you can show you can't have a constant additive error. For a $+c$-spanner, if you have a weighted graph, you can use the same example of a complete bipartite graph (now the edges are undirected). But now you have really big weights (say M). Now if I omit any edge, the shortest path you could have now becomes $3M$ instead of M. So the additive error is proportional to your edge weights — we went from M to $3M$ when we omitted a single edge. So the additive $+c$ is actually $+\Omega(M)$ — you can't have a constant additive error (your error will be proportional to the maximum edge weight).

This is why in the additive case we care about unweighted graphs, so that we can focus on constant error. (In the weighted graphs case, you'd measure additive error in terms of the maximum edge weight; you can do that with most of these constructions, but we won't.)

Next, do you see some weirdness going on with the multiplicative vs. additive spanners? For the multiplicative spanners, for *every* integer k, you can get sparser and sparser graphs by increasing the multiplicative error. But in the additive case, we only have these three results. Why can't we get $+8$ with less than $n^{4/3}$ edges? For the longest time, people thought we *should* be able to get an additive approximation scheme (where you get sparser and sparser subgraphs with bigger additive error). But it turns out this is actually not possible in the additive regime!

Theorem 13.5

There is a family of n-node graphs such that any subgraph on $O(n^{4/3-\varepsilon})$ edges distorts some distance by $+n^{\delta}$ (where δ is some function of ε).

So the point is you cannot get a constant additive spanner, the moment you have fewer than $n^{4/3}$ edges. This is pretty crazy, but the way it's proven was as follows. Virginia had a colleague who worked on spanners, and in fact she proved the $+4$ result; and she said she doesn't knwo why we can't do better than $n^{4/3}$ she

thinks there's some barrier here. Virginia thought about it and thought we'd have to use some additive combinatorics to prove this. So she talked to her students and gave them some ideas. They disappeared for a few months, and a few months later they came and said they'd proved it.

They actually did use additive combinatorics, slightly changing what Virginia was thinking about, and created a very beautiful construction of these graphs. These graphs have roughly $n^{4/3}$ edges, and if you only leave $n^{4/3-\varepsilon}$ edges in your graph, you definitely distort some distance a lot. This won the best student paper award and so on, and kind of finished off this line of research for additive approximations, with one caveat.

Now we know you can't decrease $n^{4/3}$ — so this is basically the best you can do for the sparsity. We also know from another result that if you want +2 error, you need at least $n^{1.5}$ edges, so this is also optimal. The only thing you could potentially gain is maybe you could get a +4 spanner in $n^{4/3}$ edges — we don't know.

Another weirdness about this thing — all these +2, +4, and +6 are even. Why? (It's the same with additive APSP.) Why? It's because basically the difficult graphs are bipartite graphs. And in bipartite graphs, if you omit an edge, then any detour adds an additive even error. This is roughly why — bipartite graphs are the tough objects here. So any sort of existential result that talks about every graph will have an additive even error.

We're going to prove all three results in Theorem 13.3, and in the end we'll talk a bit about Theorem 13.4 (which turns out to be much simpler, which is why it was known before).

§13.3 A +2 additive spanner

We'll start with +2. This will be analogous to the +2 APSP algorithm we gave last time — last time we gave a +2 APSP warmup, which had runtime $\widetilde{O}(n^{2.5})$. That will basically imply this result.

So first we'll show that there's a +2 additive spanner on $\widetilde{O}(n^{1.5})$ edges. In all our sanner results, we'll say your input is some undirected $G = (V, E)$; and for every pair of vertices, we'll use P_{uv} to denote some $u \rightsquigarrow v$ shortest path. (There could be multiple shortest paths, but we just focus on one of them — think about this as a fixed shortest path. We're not providing efficient algorithms, so assume someone gave it to us.)

How does the +2 APSP algorithm work? We say *low-degree* vertices have degree less than $\sqrt{n}$, and *high-degree* vertices have greater degree. We take a hitting set S of size $\sqrt{n}\log n$ and say it hits the neighborhoods of high-degree vertices. Last time, we said that if we remove the high-degree vertices, then the remaining graph has $n\sqrt{n}$ edges.

What we're going to do here to build our spanner $H = (V, E_H)$ is we'll start with E_H being empty, and we'll keep adding edges.

First, for every low-degree vertex v, we'll add *all* its edges to the spanner. So we add all edges incident to low-degree vertices. This will give us $n \cdot \sqrt{n}$ edges. And we also get that every P_{uv} with no high-degree vertices is in H, meaning that $d(u, v) = d_H(u, v)$.

Next, we have this hitting set. What we did before (for +2 APSP) was we ran BFS from every node $s \in S$, and we set $d'(u, v) = \min_{s\in S}(d(u, s) + d(s, v))$. If we have u and v and the path has a high-degree node x, then it has some neighbor $s \in S$; and then when we ran BFS, we have a shortest path from u to s and s to v; so we get

$$\min_s(d(u, s) + d(s, v)) \leq (d(u, x) + 1) + (1 + d(x, v)) = 2 + d(u, v).$$

• •

Now, instead of computing the *distance*, what we'll do is that after we've run $\mathsf{BFS}(s)$, we'll have computed the BFS tree at s (the shortest path tree rooted at s). This shortest path tree has at most $n - 1$ edges,

because it's a tree; we call it T_s. Then we add all edges of T_s to E_H. This means the edges on the shortest path (for every u and s) are all in H; and because of this, we get that

$$d_H(u,v) = \min_{s \in S}(d(u,s) + d(s,v)) \leq 2 + d(u,v)$$

when P_{uv} has a high-degree node.

And we added $n-1$ edges for every $s \in S$, so the number of edges added is at most $|S| \cdot (n-1) \leq \widetilde{O}(n^{1.5})$ (because $|S| \sim \sqrt{n} \log n$).

So both of these are roughly $n^{1.5}$; so with $n^{1.5}$ edges we have achieved a $+2$ approximation in the spanner.

If P_{uv} has no high-degree vertices, then the entire shortest path is in the spanner, so the distance is exact. If it does have a high-degree vertex, then from its neighbor s in the hitting set we have its entire shortest paths tree, so we have a shortest path from u to s and s to v, and by what we saw last class (the triangle inequality), we know this distance is at most $d(u,v)+2$. So the previous algorithm immediately gives us a spanner, just by adding the correct edges.

Remark 13.6. Last time, we also saw a *faster* algorithm that gave $+2$ APSP — a $n^{7/3}$ algorithm. You might think that $n^{2.5}/n = n^{1.5}$, so maybe if I divide $n^{7/3}/n = n^{4/3}$ I might get a $+2$ spanner with $n^{4/3}$ edges. But we just said that $n^{1.5}$ is best possible for $+2$. So why can't we turn that second algorithm into a spanner?

Most of that algorithm works fine; but the issue is that in that algorithm, in the last step, for every u we created a graph G_u. And we added some subgraph of G, but we also added $n^{2/3}$ weighted edges for every u. We could attempt to merge all these graphs together, but now we have $n \cdot n^{2/3} = n^{5/3}$ weighted edges, which is much more than $n^{4/3}$. So each G_u has $n^{4/3}$ edges, but we can't combine them into a single graph with $n^{4/3}$ edges. So the problem is that we can't combine them — first because they're new weighted edges, and also because there's too many. (And we know this is necessary, because $n^{1.5}$ is the best we can do for $+2$.) So there is a difference between additive APSP and additive spanners.

§13.4 A new tool — neighbors of paths

Next we'll talk about the $+4$ and $+6$ additive spanners. These have a new tool.

For $+2$, what we used was that the shortest path between u and v has *some* high-degree vertex. But it turns out that if it has *many* high-degree vertices, then we can gain something.

Imagine L and D are parameters which are some $\mathsf{poly}(n)$ (e.g., $\sqrt{n}$ or $n^{1/3}$ or something).

Lemma 13.7

If a shortest path P_{uv} has at least L vertices of degree at leats D, then P_{uv} has $\Omega(L \cdot D)$ neighbors.

So you have your path; and if it has L high-degree vertices, then the number of nodes that have a neighbor on the path has to be more than LD.

Why is this useful? Imagine I wanted to have an additive 2-approximation, but only cared about paths with at least L neighbors. Then it's easier to hit the neighbors with a hitting set. Before we only guaranteed there was one high-degree vertex, so we used a hitting set of size $\frac{n}{D} \log n$. But if this lemma is true, then we can use a hitting set of size $\frac{n}{LD} \log n$ instead. And then I'll find some neighbor of this path in that hitting set. And this is smaller, because we have an extra L in the denominator.

When we cared about $+2$-approximations, all we cared about was that we find a neighbor of some node on the path. So if we can do this with a smaller hitting set, then I can add $\frac{n}{LD} \log n$ shortest path trees, rather than $\frac{n}{D} \log n$, to get a $+2$-approximation for $d(u,v)$.

We'll prove this lemma, and then use it to get the $+4$ and $+6$ additive spanners. To prove it, here's a claim.

Claim 13.8 — Let P_{uv} be any shortest path. Then for any $x \in V$, x has at most 3 neighbors on P_{uv} (if $x \in P_{uv}$, then it has at most 2).

What is this saying? Imagine we have our path P_{uv}, and there's some node x. And we're claiming it can't have more than 3 neighbors on the path.

Proof. Suppose for contradiction that it has 4 neighbors.

Let's call them y_1, y_2, y_3, y_4 in that order. Why can't this happen? Well, P_{uv} was supposed to be a shortest path. But I could instead take the path $u \to y_1 x y_4 \to v$. And this is shorter — because $y_1 \to y_2 \to y_3 \to y_4$ on the original path was at least 3, but x gives a way of getting from y_1 to y_4 in just two edges. And this is a contradiction — P_{uv} was supposed to be a shortest path, but it's not.

(Similarly, if x were on the shortest path, then you can omit one of these and run the same argument to say there's at most 2.) □

Proof of Lemma. Suppose P_{uv} has a lot of high-degree nodes — at least L nodes of degree at least D.

We're going to measure the number of *edges* from P_{uv} to outside P_{uv} — there's L of these nodes, and for each there's at least $D-2$ edges that aren't part of the path. So the number of these edges going out of the path is at least $L(D-2)$.

Now let's consider the set of neighbors (which are not part of P_{uv}). Every one of these has at most 3 neighbors on the path — that means in the yellow set of edges, each of the vertices in the green thing has yellow degree at most 3. So this means by an averaging argument, the number of (yellow) neighbors of the path is at least $L(D-2)/3 = \Omega(LD)$. □

§13.5 A +4 additive spanner

Now we have the tools to do the +4 spanner. We have an undirected graph $G = (V, E)$, and we're going to be building up a spanner $H = (V, E_H)$; and again, we assume we have these shortest paths P_{uv}.

We're going to pick parameters L and D later.

The first thing we'll do, just like with the +2 spanner, is that we'll add to E_H all edges incident to nodes of degree at most D (we think of these as *low-degree*). This gives us nD edges; and it also gives that if P_{uv} has no high-degree nodes, then $d_H(u,v) = d(u,v)$.

Next, very similarly to before, we create T with $|S| \sim O(\frac{n}{D} \log n)$ to be a hitting set for (the neighborhoods of) high-degree nodes.

And we also create T with $|T| \sim O(\frac{n}{DL} \log n)$ to be a hitting set for *paths* with at least L high degree nodes.

With T, we'll do what we did with the +2 spanner — wee'll add to E_H the shortest path tree rooted at every $t \in T$. (This is just like before, except now we're doing it from a smaller subset.) Then we get that the number of edges we added is the number of nodes in T, times n — so we add n^2/DL edges. And we also have that if P_{uv} has at least L high-degree nodes, then we have $d_H(u,v) \le d(u,v) + 2$ — so we already

got a +2-approximation if it so happens that P_{uv} has a lot of high-degree nodes (because the hitting set will hit its neighborhood, and it's the same reasoning as before).

What shortest paths are *not* well-approximated by this spanner so far? The ones that have high-degree vertices, but fewer than L of them. So the only shortest paths that remain (that are potentially not approximated well) are the ones with

$$\#\text{high-degree nodes} \in [1, L).$$

So here, we'll use something else. Imagine you have u and v, and we know it has at most L high-degree vertices. And let x be the *first* high-degree vertex, and y the *last* one.

Now, we know that there exists some s in our hitting set S that's a neighbor of x; we call it s_x. Similarly, there exists some $s_y \in S$ which is a neighbor of y.

What we'll do is kind of similar to what we did in the additive APSP — for every high-degree vertex, we'll artificially add an edge to something in S to the spanner. So for every high-degree node v, we add some (v, s) where $s \in S$ is a neighbor of v to E_H. (Every high-degree vertex has a neighbor in S; we arbitrarily pick one such neighbor and add an edge.) This only adds n edges, but it'll be helpful.

Now what we're going to try to do is add some path connecting s_x to s_y, and add it to the spanner. What we *do* know is that this path between x and y has less than L high-degree nodes. So in particular, at most L edges on this path are missing from H — the only possible edges missing from H are those between two high-degree vertices on the path (because if your edge has an endpoint that's low-degree, we've already added it to H).

So this path from x to y (in P_{uv}) has at most L edges missing from H. If we could somehow get a handle on this path and add just the L edges that are missing, we'd be in very good shape.

And we can basically do this as follows: For every pair $s, s' \in S$ (our hitting set for the high-degree vertices), we consider all possible paths between s and s' that are missing at most L edges from H. Let $Q_{ss'}$ be the shortest such path, and add its at most L missing edges to H.

So we're adding at most L missing edges for every $s, s' \in S$; this means the number of edges we add here is the number of pairs of nodes in S (which is $(n/D)^2$) times the L missing edges (the number of edges we added), which is $\widetilde{O}((n/D)^2 \cdot L)$.

Now we have to show that this is actually good enough for us. This completes the description of the spanner. What we need to show is this last step handles paths whose number of high-degree nodes is between 1 and L; and we also have to analyze the number of edges in the spanner.

Consider some P_{uv} with less than L high-degree nodes; and say x is the first and y is the last high-degree node, and you have your s_x and s_y.

Now, the edge $s_x x$ is in the spanner; everything from u to x and from y to v is in the spanner (because they only have low-degree vertices); and $y s_y$ is also in the spanner.

Now when we considered $(s, s') = (s_x, s_y)$, we looked at all possible paths between s_x and s_y that are missing at most L edges. And we claim the path $s_x x \to y s_y$ is missing at most L edges. That's because the portion $x \to y$ is missing at most L edges (there's at most L high-degree nodes, and at most one per high-degree node is missing). And $s_x x$ and $y s_y$ are in the spanner.

And in our procedure, we picked the *shortest* such path $Q_{s_x s_y}$ — so we picked some path $Q_{s_x s_y}$, and that's the shortest, so its length is at most the length of the path $s_x x \to y s_y$. So

$$\left|Q_{s_x s_y}\right| \leq 1 + d(x,y) + 1 = 2 + d(x,y).$$

So in H, we have that

$$d_H(u,v) \leq (d(u,x)+1) + \left|Q_{s_x s_y}\right| + (1 + d(y,v)) \leq 4 + d(u,x) + d(x,y) + d(y,v) = 4 + d(u,v)$$

(where $d(u,x)+1$ is the first green part — u to x to s_x).

So it's as if — if this yellow path were actually this one, it'd correspond to going $u \to x \to s_x \to x \to y \to s_y \to y \to v$. (But it might be something shorter.)

Now we want to argue about the size of the spanner. First we added nD edges for the low-degree vertices; then n^2/DL for the shortest path trees; then we added n (for the edges (v,s) for high-degree v); and then we added $(n/D)^2 \cdot L$. The n is subsumed by nD, so we can just kind of forget it.

We want to minimize this, so let's set $nD = n^2/DL$. From here we get $D^2 L = n$, so $L = n/D^2$.

Then we should set the last two terms equal; this gives $nD = (n/D)^2 L = (n/D)^2 \cdot n/D^2$, and from here you get $D^5 = n^2$, so $D = n^{2/5}$; and so nD becomes $\widetilde{O}(n^{7/5})$.

So we get $n^{7/5}$-edge spanners with additive error at most 4.

§13.6 A +6 additive spanner

Next we'll do the +6 spanner. As before, we'll have a parameter D, and add to E_H all edges incident to nodes of degree less than D; so if P_{uv} has no high-degree nodes, then $d_H(u,v) = d(u,v)$. As before, we'll take S to be a hitting set of the high-degree nodes.

Now for all $i \in \{0, \ldots, \log n\}$, we'll create a hitting set S_i of

$$|S_i| \sim \frac{n}{D \cdot 2^i} \log n$$

(where $S_0 = S$). So we have $\log n$ hitting sets; and S_i hits the neighborhood of any P_{uv} with

$$\#\text{high-degree nodes} \in [2^i, 2^{i+1})$$

(in fact, they hit the neighborhoods of every P_{uv} with at least 2^i high-degree nodes).

So now we have $\log n$ hitting sets.

Suppose P_{uv} has #high-degree nodes $\in [2^i, 2^{i+1})$. So you have u and v; and let x be the first high-degree node and y the last one

Just like before, for every high-degree node v, we add some (v,s) edge to E_H (where $s \in S$). So just like before, x will have some neighbor $s_x \in S$, and y will have some neighbor $s_y \in S$.

We also know that there exists some neighbor t of P_{uv} which is in S_i (because P_{uv} has enough high-degree nodes, so we know S_i hits it).

Now what we're going to try to do is to try to connect s_x to t. How do I do this? Well, I want to add a path, just like we did before, between s_x and t, as long as it's missing at most 2^{i+1} edges.

So what we know here is that the number of high-degree nodes on P_{uv} is at most 2^{i+1}. So the portion of it from x to y is missing at most 2^{i+1} edges from H. So we're going to try to add a path from s_x to t that's missing at most 2^{i+1} edges from H.

The issue right now is in the previous argument I assumed the edge to t was in H. But here, this edge is not necessarily in H (and it's unclear how to force that). One way to force that is to say for every vertex that's high-degree, if it has some neighbor in S_i, add that edge. So for every high-degree v, if v has some neighbor $t \in S_i$, then add some edge (v, t) to H. (We pick an arbitrary edge, but only if v has a neighbor in S_i. Some high-degree nodes might not have neighbors in S_i.) Here we're doing this for every i, so we add $O(n \log n)$ edges.

So now we can assume that there is some node z on this path that has high degree and has a neighbor $t \in T$.

Now we can do exactly what we did before, with a slight change. So for every $s \in S$ and every $t \in S_i$, we look at all paths from s to t which are missing at most 2^{i+1} edges from H, and let $Q_{st}^{(i)}$ be the shortest out of them. And we add the missing edges of $Q_{st}^{(i)}$ (of which there are at most 2^{i+1}) to H.

So we have s and we have t, and we look at all possible paths that are missing less than 2^{i+1} edges; and we pick the shortest one, and that's the one we add.

How many edges do we add to the spanner from this procedure? It's

$$|S| \cdot |S_i| \cdot 2^{i+1} \approx \frac{n}{D} \cdot \frac{n}{D \cdot 2^i} \cdot 2^{i+1} \approx \frac{n^2}{D^2}$$

(omitting log factors). And if we do this for all i, we get a $\log n$ factor (because there's only $\log n$ choices for i). So I actually get to add only n^2/D^2 edges; and this means the spanner will be sparse.

So what's the size of the spanner here? It's nD for the low-degree vertices; and then we get $(n/D)^2$ from this. So now when we set these to be equal we get $D^3 = n$, so $D = n^{1/3}$; and the size of the spanner is actually $n^{4/3}$.

But now we have to say why it works (i.e., why this is a $+6$ spanner); this will be very similar to what we did for the $+4$ spanner.

To show this, let's look at any shortest path from u to v; for some i, it'll have exactly between 2^i and 2^{i+1} high-degree nodes (if it has no high-degree nodes, then we've already handled it). So there's u and v, and we have this picture:

When we created the spanner, we know $u \to x$, xs_x, $x't$, ys_y, and $y \to v$ are all in the spanner. Now when we ran this algorithm, for (s_x, t) we added a shortest path missing at most 2^{i+1} edges. And notably, the part $s_x x \to x't$ is missing at most 2^{i+1} edges; similarly, $tx' \to ys_y$ is also missing 2^{i+1} edges. And in the spanner, we added some path $Q_{s_x t}^{(i)}$ and some $Q_{ts_y}^{(i)}$; and just like before, we get that

$$\left|Q_{s_x t}^{(i)}\right| \leq 1 + d(x, x') + 1 = d(x, x') + 2$$

(because this was a viable path, and $Q_{s_x t}^{(i)}$ was the *shortest* viable path), and similarly

$$\left|Q_{ts_y}^{(i)}\right| \leq d(x', y) + 2.$$

And therefore, we have

$$d_H(u, v) \leq d(u, x) + 1 + \left|Q_{s_x t}^{(i)}\right| + \left|Q_{ts_y}^{(1)}\right| + 1 + d(y, v) \leq 6 + d(u, v).$$

This kind of corresponds to going $u \to xs_x x \to x'tx' \to ys_y y \to v$.

So this is the sparsest known additive spanner, and we don't know whether you can improve from $+6$ to $+4$.

Student Question. *Is it not kind of strange that we have this property on the S_i that there's so many high-degree nodes in each S_i that are adjacent to the path, but we only use one of them?*

Answer. We're only guaranteed that there's at least one that's a neighbor of the path — we're saving a 2^i in the size, but that means all we can guarantee is there's at least one.

§13.7 Multiplicative spanners

We won't prove Theorem 13.4 today — we'll prove it next time — but we'll say a few words.

In the next two minutes, we'll give the algorithm that produces a multiplicative spanner for every k:

Algorithm 13.9

Initialize $E_H \leftarrow \emptyset$.

- Sort the edges of G in nondecreasing order of weight:
- For each (u, v) (processed in nondecreasing order of weight):
  - If $d_H(u, v) > (2k-1) \cdot w(u, v)$, add (u, v) to E_H.

So that's it. There's two claims.

Claim 13.10 — This graph H is a $(2k-1)$-spanner.

What does this mean? Consider any shortest path P_{uv}, and suppose some edge xy is missing from this path — so it's not in E_H. But why didn't we add it? We didn't add it because the distance in the spanner was already at most $2k-1$ times its weight. So in the spanner, there is a path with $d_H(x, y) \leq (2k-1)w(x, y)$. And this is true for every edge on the path — either it's in the spanner, or you can detour it by something that's at most a factor of $2k-1$ more. So in the spanner, $d_H(u, v)$ is at most a $(2k-1)$-factor off from what it should be — we have

$$d_H(u, v) \leq (2k-1) \sum_{xy \in P_{uv}} w(x, y) = (2k-1)d(u, v).$$

So it's clearly a $(2k-1)$-spanner; all you have to show is that the number of edges is very small. We'll do this next time. The reason why it's small is that this graph H you're creating does not have any short cycles — specifically, if you ignore the weights, it has no cycles of length at most $2k$. That's because if you assume for contradiction that there is a cycle of length at most $2k$ and you look at the largest edge weight on that cycle (call it e), then in this algorithm it was considered last (because we considered them in nondecreasing order). But because it's considered last and it has largest edge weight, it was added because $d_H(u, v) > (2k-1)w(u, v)$ at that time. But that gives you a contradiction — the cycle has at most $2k$ edges, so omitting this edge, on that cycle there's a path with $2k-1$ edges, each of which has weight less than $w(e)$. So that means there was already a path of length at most $(2k-1)w(e)$, and so $d_H(u, v) \leq (2k-1)w(u, v)$. So this edge would actually not have been added, because the condition for adding it would have failed.

So there's no way this edge could be added, which means there's no way you could close a cycle — any cycle you create has length bigger than $2k$. And next class, we'll show that any graph with no short cycles is actually sparse. (That's a simple argument.)

§14 April 1, 2025 — Distance Oracles

Hopefully spring break went well. Today we're going to continue a bit about spanners, and then move on to distance oracles.

§14.1 Review — multiplicative spanners

As a recap, last class we discussed additive spanners and multiplicative spanners. Today multiplicative spanners will be relevant:

Definition 14.1. A *α-spanner* of $G = (V, E)$ is a subgraph H such that

$$d(u, v) \leq d_H(u, v) \leq \alpha \cdot d(u, v) \quad \text{for all } u, v \in V.$$

We always have $d_H(u, v) \geq d(u, v)$ (because H is a subgraph).

Last class we showed that every undirected graph contains a spanner which is pretty sparse.

Theorem 14.2

For all integers $k \geq 1$, every undirected graph $G = (V, E)$ on n nodes with nonnegative weights has a $(2k-1)$-spanner on $O(n^{1+1/k})$ edges.

We did this by a simple greedy algorithm — we sorted the edges by weight, and checked for every edge whether it should be added to the spanner — and we do that when the distance between its two endpoints is not already good enough. This gives a $(2k-1)$-spanner; so it remains to show the number of edges is small.

The way we showed that was by showing that the graph we constructed cannot have a cycle of length at most $2k$ (ignoring weights). This is because if we looked at the last edge in that cycle added by the algorithm, then the path formed by the rest of the cycle is already of length $2k-1$, so we wouldn't have added that edge (and this works even with weights).

Then we said that if your graph doesn't have chycles of length at most $2k$, then the number of edges can't be more than e.g. $10n^{1+1/k}$. The reason for that is first you can greedily remove vertices whose degree is very small (e.g., less than $9n^{1/k}$). When you remove those, you've removed at most $n^{1+1/k}$ edges. If at this point the graph is empty, then you're done. Otherwise, you end up with some subgraph where every vertex has very high degree — let's say more than $9n^{1/k}$. And if your graph is very dense and every vertex has degree more than e.g. $9n^{1/k}$, and you start a BFS from an arbitrary vertex and go k levels, then because the graph supposedly doesn't have cycles of length at most $2k$, you'll never visit a vertex twice. So up to level k, it just looks like a tree. But every vertex in the tree has downwards degree more than $n^{1/k}$. And after k levels, you'll have visited more than n vertices. This is a contradiction, because the graph has n vertices.

So either you visit a vertex more than once, then you've found a cycle of length at most $2k$; or you've visited more than n vertices, which is a contradiction. So any dense enough graph contains a cycle of length at most $2k$; and that means this algorithm must create a very sparse spanner.

Now, there's this thing called teh Erdős girth conjecture. What we do is:

Definition 14.3. We define $m_k(n)$ as the maximum number of edges of an n-node undirected graph of girth at least $2k+2$.

It's really girth greater than $2k$; but if you have a graph of girth $2k+1$ you can make it bipartite, and then its girth becomes at least $2k+2$.

Conjecture 14.4 (Erdős girth conjecture) — We have $m_k(n) \geq \Omega(n^{1+1/k})$.

We showed above that if your graph has at least $10n^{1+1/k}$ edges then it must have girth at most $2k$ (the 10 is loose — you can make it smaller) — so this shows $m_k(n) = O(n^{1+1/k})$. The girth conjecture says that this is tight.

So the girth conjecture says that for every k, there exists some n-node graph of girth at least $2k+2$ whose number of edges is at least $n^{1+1/k}$ — so for every k, there are dense graphs which have high girth. This conjecture has been proven for very small k — for $k = 1$, 2, 3, and 5. (We don't know it for 4, and we don't know it for $k > 5$; that's why it's still a conjecture.)

Now what Virginia will tell us is that if you believe the EGC, then this sparsity bound on multiplicative spanners is optimal — you cannot get sparser $(2k-1)$-spanners.

Claim 14.5 — Under the Erdős girth conjecture, for every integer $k \geq 1$, there exists an n-node graph such that all its $(2k-1)$-spanners have at least $m_k(n) \geq \Omega(n^{1+1/k})$ edges.

Proof. For some notation, we'll say a *EGC graph* is an n-node G with $\Omega(n^{1+1/k})$ edges and girth at least $2k+2$. The conjecture says that for every k, such a graph must exist.

Assuming the EGC, it would be natural to pick G to be one of these EGC graphs. So let G be an EGC graph. Now, we claim that any $(2k-1)$-spanner of G must be G itself. Why? Suppose for contradiction that H is missing some edge of G — so $(u,v) \in G \setminus H$. Then we must have $d_H(u,v) \leq (2k-1)d(u,v) = 2k-1$, so in H you have a path from u to v of length at most $2k-1$, and this path doesn't contain the edge uv. So together in the graph G, you have a cycle (consisting of this edge and the path) of length at most $2k$. But G didn't have $2k$-cycles — its girth was at least $2k+2 > 2k$ — so we get a contradiction. So that means H is just G — it's not missing any edges. □

This means if you believe the EGC, the theorem we proved last time is optimal — you cannot get fewer edges for every graph.

§14.2 Distance oracles

Now we're going to define a generalization of spanners — something called *distance oracles* — and we'll see what we can do with them. An α-distance oracle is really a data structure defined by two algorithms — one is a *preprocessing* algorithm, and the other is a *query* algorithm. The preprocessing algorithm takes in a graph and produces a 'summary' of the graph which takes less space. And for the query algorithm, you give it two vertices $u, v \in V$, and it needs to output an estimate of the distance between u and v. But it doesn't get to see G — what it sees is only the summary.

Definition 14.6. An *α-distance oracle* (α-DO) consists of two algorithms:

- A *preprocessing* algorithm, which takes in $G = (V, E)$ (which is undirected and has nonnegative weights) and produces a *summary* $S(G)$.
- A *query* algorithm, which takes in $u, v \in V$; and given only access to $S(G)$ (not G itself), it outputs some estimate $D(u,v)$ such that $d(u,v) \leq D(u,v) \leq \alpha \cdot d(u,v)$.

The quality of a distance oracle is determined by four things:

- The approximation factor α. You want this to be as close to 1 as possible.
- The space usage, i.e., $|S(G)|$.
- The query time (which is hopefully $O(1)$).
- The preprocessing time. We're going to ignore this part for this lecture and not worry about it too much; you might worry about it on your problem set. But the first three are the most important things.

§14.3 Spanners as distance oracles

First, we claim that we can already get a distance oracle using a $(2k-1)$-spanner (as the distance oracle). What's the approximation factor? Well, it's $2k-1$. The space usage is $O(n^{1+1/k})$ words (since you have to store its edges). And what's the query time? This is also $\widetilde{O}(n^{1+1/k})$, since you can run Djikstra's from u to figure out the distance to v. But this is not very good — we actually want this to be a constant, but unless you really change what you're doing, the best thing you can do is to run Djikstra's.

But even though the query time is kind of bad, we're going to show that this space usage of $O(n^{1+1/k})$ is basically optimal under the Erdős girth conjecture. So in some sense, these spanners are pretty good.

Claim 14.7 — Assuming the EGC, for any α-DO with $\alpha \leq 2k$, there exists an n-node G whose space usage $|S(G)|$ is at least $m_k(n) \geq \Omega(n^{1+1/k})$ bits.

So that means (up to the 'words' vs. 'bits' distinction), the space usage of the spanners are optimal.

Proof. Let G^* be an EGC graph with $m_k(n) \geq \Omega(n^{1+1/k})$ edges and girth at least $2k+2$. What we're going to do is take this graph, and consider all its possible subgraphs. How many subgraphs does G^* have? A subgraph means some graph with the same vertices, but some subset of the edges; so the number of subgraphs is $2^{m_k(n)}$. So there's a lot of these subgraphs. What we're going to show is that every pair of these subgraphs must have distinct storages under the distance oracle — there canot be two different subgraphs that have the same storage.

Suppose for contradiction that $H_1 \neq H_2$ are subgraphs of G^* and their storages under the distance oracle are the same, i.e., $S(H_1) = S(H_2)$. Because these guys are different, one of them contains an edge that the other one doesn't; so without loss of generality, there is some edge uv that is in H_1, and uv is not in H_2. (If it's the other way around, we just swap the roles of H_1 and H_2.) This means $d_{H_1}(u,v) = 1$. And $d_{H_2}(u,v)$ is not 1; but moreover, it is at least $2k+1$ (because G^* has girth at least $2k+2$) — if it were smaller, then we'd have a cycle of length less than $2k+2$ in G^*.

So the distance in H_2 is really large, while the distance in H_2 is exactly 1.

And the distance oracle supposedly returned α-approximate distances — so let's say in H_1, it returns $D_{H_1}(u,v)$. Since the actual distance is 1 and it returns an α-approximation (where $\alpha \leq 2k$), it has to return some

$$D_{H_1}(u,v) \leq \alpha \cdot 1 \leq 2k.$$

And on H_2, the distance oracle returns

$$D_{H_2}(u,v) \geq d_{H_2}(u,v) \geq 2k+1$$

(since the estimate it returns is always at least the true distance). Notably, this is bigger than $D_{H_1}(u,v)$.

And this already gives us a contradiction — the query algorithm of the distance oracle only has access to the summary $S(G)$. And we assumed the summaries of H_1 and H_2 were equal. There's no way the distance oracle takes two summaries that are exactly the same ($S(H_1) = S(H_2)$) and two inputs that are exactly the same (u and v) and returns different outputs. So this is a contradiction.

This means every pair of different subgraphs of G^* must have different summaries. So there are $2^{m_k(n)}$ different summaries, which means the number of bits needed to describe each summary is $m_k(n)$ (which is $\Omega(n^{1+1/k})$ under the EGC). □

So not only is the spanner size optimal (under the EGC), but also any *distance oracle* must take a lot of space — even though distance oracles can store any sort of information. They don't need to store a graph — they can store whatever else you want — but even then you still need this much space.

Student Question. *Do people know distance oracles that have more efficient query time — can you get the same space but faster queries?*

Answer. This is what we're going to do next!

This seems optimal in terms of space, but its running time is awful. So we're going to fix the runtime and make it constant; that's the next part of the lecture.

§14.4 Warmup — a 3-approximate distance oracle

What we're going to do first is a warmup to how to get distance oracles — we're going to give a 3-approximate distance oracle with $\widetilde{O}(n^{3/2})$ space and $O(1)$ query time. It'll be very simple, and we're not going to worry about the preprocessing time. This is the case $k = 2$ (since $2 \cdot 2 - 1 = 3$); and once we get that, we'll generalize it to bigger k.

So our input is an undirected graph $G = (V, E)$ with nonnegative weights. We're going to start with a hitting set of size roughly $\sqrt{n}$ — let A be a hitting set. What does it hit? It's going to hit the closest $\sqrt{n}$ vertices to every node. And its size is $|A| = \Theta(\sqrt{n} \log n)$ (as usual; it exists, and if you have the $\sqrt{n}$ vertices closest to every node then you can find it deterministically, or you can just pick it randomly and it'll be correct with high probability).

So now we have this A. And now, for every vertex v in the graph, we find the closest vertex in A to v; so for every $v \in V$, we define $p(v)$ as the closest vertex of A to v. (The letter p stands for *pivot*; we'll see why we care about pivots, not in this algorithm but in the next.)

Now we define $A(v)$ as all the vertices that are strictly closer to v than to the pivot, i.e.,

$$A(v) = \{x \in V \mid d(v, x) < d(v, p(v))\}.$$

What is this? We have $p(v)$ being the closest vertex of A to v; and $A(v)$ is all the vertices strictly closer to v than $p(v)$ is. These definitions mean $|A(v)| < \sqrt{n}$ — this is because A is a hitting set of the closest $\sqrt{n}$ vertices of every v, so $p(v)$ is among the closest $\sqrt{n}$ vertices to v, and therefore all the vertices closer to v must have number less than $\sqrt{n}$.

And then we're going to define something called the *bunch* (some people call it the *ball*)

$$B(v) = A(v) \cup A.$$

Clearly $|B(v)| = O(\sqrt{n} \log n)$, because $|A| = O(\sqrt{n} \log n)$ and $|A(v)| = O(\sqrt{n})$.

Now we've defined all this stuff, and we're going to say the summary of the graph. The summary $S(G)$ is: For every $v \in V$, I'm going to store $p(v)$. And also, for every $x \in B(v)$, I'm going to store $d(v, x)$. So we store the closest vertex to v in A; and for everything in the bunch ($A(v)$ together with A), we store the true distance.

Then $|S(G)|$ is the total size of the bunches, plus a little bit; so it's

$$|S(G)| = O(n^{3/2} \log n).$$

For the query algorithm, we're given $u, v \in V$. The first thing we're going to check is, is the distance between u and v stored in our summary? If $v \in B(u)$, then we have the correct distance $d(u, v)$ stored in $S(G)$, so we just return $d(u, v)$. (This is the easy case.)

Otherwise, $v \notin B(u)$. In that case, we return

$$d(u, p(u)) + d(p(u), v).$$

Claim 14.8 — Both of these distances are stored in our summary.

This is just because $p(u)$ is in A, and we included A in all the bunches — so $A \subseteq B(u)$ and $A \subseteq B(v)$. So we have both of these distances in the summary, and we can return their sum.

Now we're going to show that this is a 3-approximation (when we're in the case $v \notin B(u)$).

Claim 14.9 — If $v \notin B(u)$, then $d(u, p(u)) + d(p(u), v) \leq 3d(u, v)$.

Proof. First, what does $v \notin B(u)$ mean? This means v is not in $A(u)$ (because it's not in $B(u)$, and $B(u)$ includes $A(u)$). And now $A(u)$ was defined as all the vertices in the graph which are closer to u than its pivot was. So because $v \notin A(u)$, this implies

$$d(u, v) \geq d(u, p(u)).$$

Then we're in good shape: we can start with $d(u, p(u)) + d(p(u), v)$ and do some dancing with the triangle inequality. We have

$$d(p(u), v) \leq d(p(u), u) + d(u, v)$$

by the triangle inequality. And $d(u, p(u))$ and $d(p(u), u)$ are the same, so we get that this is at most

$$2d(u, p(u)) + d(u, v).$$

But we get $d(u, p(u)) \leq d(u, v)$ by what we said above, so this is at most $3d(u, v)$. □

So this is how you get a 3-approximate distance oracle with roughly $n^{3/2}$ space. It's very simple — you store just enough distances, and apply the triangle inequality a few times.

§14.5 A $(2k-1)$-distance oracle with constant query time

This is just a warmup; now we're going to do the full $(2k-1)$-approximate distance oracles. We'll see a construction by Thorp and Zwick from 2001, with the caveat that we're not going to care about preprocessing time (you'll care about it on the problem set). We'll have $O(1)$ query time (really $O(k)$, but k is a constant), and $\widetilde{O}(n^{1+1/k})$ space.

We're going to build on what we did before. Instead of a single hitting set, we're going to have a *hierarchy* of hitting sets. So we start off with $V = A_0$. And then inside A_0, we'll pick something called A_1, which will be a hitting set of something. And so on; so we'll have

$$V = A_0 \supseteq A_1 \supseteq \cdots \supseteq A_{k-1}.$$

What is A_i? For every i, we take A_i to be a hitting set for the $n^{1/k}$ closest vertices in A_{i-1} to v, for every v.

So here's a picture: Imagine we draw v somewhere in the outermost circle, and let's say we have A_{i-1} (one of the intermediate circles), and then A_i immediately inside that. So we find the closest $n^{1/k}$ vertices to v in A_{i-1}. And A_i hits this in some vertex. So for every vertex, we have some node of A_i hitting the closest $n^{1/k}$ vertices in A_{i-1}. (We don't care about the running time here — if we cared about runtime and a deterministic algorithm, naively we'd have to compute the closest $n^{1/k}$ nodes to each v and find a hitting set.)

How large is this? We'll have

$$|A_i| \leq O\left(\frac{|A_{i-1}|}{n^{1/k}} \cdot \log n\right).$$

Basically, you have a universe of size $|A_{i-1}|$, and you only care about n subsets of size $n^{1/k}$, and you want to hit all of them. All of these sets are inside A_{i-1}, so the size of your hitting set A_i is going to be this.

So if you apply some sort of reasoning and assume k is a constant, you get that

$$|A_i| = \widetilde{O}(n^{1-i/k}).$$

(There is some $(\log n)^i$ factor, but i is a constant.) And notably, we care about the last set A_{k-1}; we have

$$|A_{k-1}| = \widetilde{O}(n^{1/k})$$

(because $i = k-1$, so we get $1-(k-1)/k = 1/k$).

Now we're going to define the summary. But before we do that, we're going to extend these pivots that we had before. So for every v and i, let $p_i(v)$ be the closest node of A_i to v. So we have some v, and then you have your A_i; so we find the closest node of A_i, and we call that $p_i(v)$. So we draw A_{i-1}, and inside that we draw A_i, and somewhere inside A_i we draw our pivot $p_i(v)$.

Then very similarly to before, we define

$$A_i(v) = \{x \in A_i \mid d(v,x) < d(v, p_{i+1}(v))\}.$$

So if I change the indices in this picture to i and $i+1$ (instead of $i-1$ and i), I look at all the nodes in A_i that are strictly closer to v than $p_{i+1}(v)$ is. These are the nodes at the bottom of the picture, so that's $A_i(v)$. And just as in our warmup, we know $|A_i(v)| < n^{1/k}$ — because A_{i+1} was a hitting set of the closest $n^{1/k}$ vertices in A_i, and these guys are strictly closer to v than any node in A_{i+1}, which means their size is bounded by $n^{1/k}$.

So if you look at our 'onion' picture (with all the A_i's drawn as nested circles), we have these pivots everywhere, and we have these A_i's at the bottom of each. And now we're going to define the bunch of v very similarly to before. In the warmup, we had a single $A_i(v)$ and a single middle hitting set. So here we're going to extend this and define

$$B(v) = A_{k-1} \cup \bigcup_{i=0}^{k-2} A_i(v).$$

And this will basically be all these green guys in the picture, together with the middle set.

And because $|A_i(v)| \le n^{1/k}$ and $|A_{k-1}| \le \widetilde{O}(n^{1/k})$, we get that $|B(v)| = \widetilde{O}(kn^{1/k})$; and k is a constant, so the factor of k disappears.

Now, the way I've picked these pivots, they're the closest node of the next layer to v. But I want to have certain consistencies, so I'm going to pick them in reverse order — I pick the closest one in A_{k-1}, then the closest in A_{k-2}, and so on; and if there's ties, I break them consistently. So we're choosing the pivots $p_i(v)$ from $i = k-1$ downwards. Suppose I've picked some pivot $p_i(v)$, and $d(p_{i-1}(v), v) = d(p_i(v), v)$. Then I'll make $p_{i-1}(v) = p_i(v)$. This ensures that if I have two pivots in adjacent layers with the same distance, then they're actually the same. (This is for technical reasons, but it makes things nice.)

> **Claim 14.10 —** For every i, we have $p_i(v) \in B(v)$.

Proof. We'll prove this by induction. First, $p_{k-1}(v)$ is in A_{k-1}, so we added it to the bunch; so it's already in the bunch, and we're done.

Now assume that $p_{i+1}(v) \in B(v)$, and consider $p_i(v)$. If $p_i(v) = p_{i+1}(v)$, then we're done. Otherwise, $p_i(v)$ is different from $p_{i+1}(v)$. And that means $p_i(v)$ is actually closer to v than $p_{i+1}(v)$ is (otherwise we would have set them to be the same) — so

$$d(v, p_i(v)) < d(v, p_{i+1}(v)).$$

But what does that mean? This means $p_i(v)$ is actually in $A_i(v)$ (because $A_i(v)$ contained all the nodes in A_i that were strictly closer to v than $p_{i+1}(v)$ is, just by definition). So $p_i(v) \in B(v)$. □

§14.5.1 The summary

So that means all the pivots are in the bunch, which is very useful. Now we'll get to what the summary and the query algorithm are.

Algorithm 14.11 (Summary)
For very v and every i, we store $p_i(v)$. And for every $x \in B(v)$, we store $d(v, x)$.

Now, this $B(v)$ consists of all these little green guys; so we store the distance from v to everything in here. In particular, we have the distances from v to all the pivots, since the pivots are in $B(v)$.

What's the size of the summary? The bunches have size $n^{1/k}$ (up to log factors), and for every node in the bunches, we're storing a distance (and we're also storing a pivot, which is less). So that means $|S(G)| = \widetilde{O}(n^{1+1/k})$ (there's n choices of v, and $n^{1/k}$ choices for $x \in B(v)$).

§14.5.2 The query algorithm

Now we'll say the query algorithm, which is a little bit crazy.

Algorithm 14.12 (Query)
Given (u, v):

- Initialize $w \leftarrow v$.
- For i from 0 to $k-1$:
  - If $w \in B(u)$, return $d(u, w) + d(w, v)$.
  - Otherwise, if $w \notin B(u)$:
    - Set $w \leftarrow p_{i+1}(u)$.
    - Swap u and v (so v becomes u and u becomes v).

Let's try to illustrate what's going on, and then we'll prove some stuff. Let's track what i is, and what the current u, v, and w are.

In the beginning (in iteration $i = 0$), w is set to v; and u is u, and v is v. Now, either we return some distance, or we move on to iteration 1. In iteration 1, u and v are swapped, and w became $p_1(u)$ (but u and v got swapped, so it's actually $p_1(v)$, for the new v).

Then either we return something, or we move to iteration 2. Now u and v are again themselves; and w becomes $p_2(v)$ (using the new u from the previous iteration).

i	new u	new v	new w
0	u	v	v
1	v	u	$p_1(u)$
2	u	v	$p_2(v)$
3	v	u	$p_3(u)$
4	u	v	$p_4(u)$

So in even iterations v and u are as they were in the beginning, and $w = p_{2j}(v)$ is the $2j$th pivot of v (notably $v = p_0(v)$). In odd iterations, u and v are swapped, so we actually have $w = p_{2j+1}(u)$. So we keep alternating between u and v.

In a picture, what's going on? Let's say we draw A_{i-1} on the outside and A_i on the inside (and u and v all the way on the outside); let's say i is odd. So we look at the pivot $p_{i-1}(v)$, and the next one will be $p_i(u)$, and so on. So we're going to keep trying to check whether this is a good estimate — e.g., $up_{i-1}(v)v$, or $up_i(u)v$, and so on. This is what we'll keep doing.

Now, we need to prove that — first, the query algorithm will return an answer, and that the answer really is a $(2k-1)$-approximation.

First, it will return an answer — in iteration $k-1$, the check $w \in B(u)$ will be true (because w is the $(k-1)$st pivot of u or v — either $p_{k-1}(u)$ or $p_{k-1}(v)$ — and this is contained in A_{k-1}, which is included in every bunch $B(u)$). So the moment we reach iteration $k-1$, we'll have $w \in B(u)$. We'll also have $w \in B(v)$, so both $d(u,w)$ and $d(w,v)$ will be inside our summary, and we can return. So this algorithm will always return an estimate.

Now what we need to show is that it will actually return a good approximation. In order to do that, we'll give two invariants, and then we'll prove them easily; and those invariants will imply what we want.

First, recall that in iteration i, we had $w = p_i(v)$ for the current v. The invariant we want is going to be:

> **Claim 14.13 —** In iteration i, we have $d(w,v) \le i \cdot d(u,v)$ (for the current w, u, and v).

Proof. We'll prove this by induction. In the base case $i = 0$, we have $d(w,v) = d(p_0(v),v) = d(v,v) = 0$ (because $p_0(v)$ is v itself).

Now suppose that it's true for iteration i, meaning that $d(p_i(v),v) \le i \cdot d(u,v)$. And also suppose that we made it to iteration $i+1$. That means we didn't return, so $w \notin B(u)$ (where w was $p_i(v)$ at that time). So this means $p_i(v) \notin B(u)$.

In particular, this means $p_i(v) \notin B(u)$, so $p_i(v) \notin A_i(u)$ (since $A_i(u)$ is inside $B(u)$). And by the definition of $A_i(u)$, this means

$$d(u,p_i(v)) \ge d(u,p_{i+1}(u)).$$

And now we can use the triangle inequality as before — we have

$$d(u,p_{i+1}(v)) \le d(u,p_i(v)) \le d(u,v) + d(v,p_i(v)).$$

And by the induction hypothesis we have $d(v,p_i(v)) \le i \cdot d(u,v)$, so we get that $d(u,p_{i+1}(u)) \le (i+1)d(u,v)$. The algorithm swapped u and v, so we swap u and v to get that $d(v,p_{i+1}(v)) \le (i+1)d(u,v)$ (for the new v, which is the old u); and this finishes the proof. □

So at iteration i, we have $d(w,v) \le i \cdot d(u,v)$.

Now we claim that this gives us the approximation quality that we want. Suppose that the query algorithm returns an estimate $d(u,w) + d(w,v)$ in iteration i; and let's look at what this is. We know that by the invariant, we have

$$d(w,v) \le i \cdot d(u,v).$$

So we have a bound on the second term. We use the triangle inequality again, so

$$d(u,w) + d(w,v) \le d(u,v) + d(v,w) + d(w,v) = d(u,v) + 2d(w,v).$$

And we said $d(w,v) \le i \cdot d(u,v)$, so we get

$$d(u,w) + d(w,v) \le (2i+1) \cdot d(u,v).$$

And now, because $i \le k-1$ (we know we will finish by iteration $k-1$), this is at most $(2k-1) \cdot d(u,v)$.

So the very worst approximation you could get is if this query algorithm keeps going up and up and only returns at the very middle of the onion; if it returns there then you get this $(2k-1)$-approximation, and if it returns at an earlier i you get a better approximation.

What's the runtime of the query algorithm? (Assume you can do constant-time lookups in your memory.) It's $O(k)$; and k is a constant, so this is $O(1)$. So you get constant query time, which is great.

So we've got pretty much optimal space usage and pretty much optimal query time.

Remark 14.14. There has been some work making this $O(k)$ into an actual $O(1)$ that doesn't depend on k — there's a way to do that.

There's also a question where the true space usage is $\widetilde{O}(kn^{1+1/k})$ and people wonder whether you can remove this k; but that's a lower-order consideration. There are also various ways to do the preprocessing fast, and you'll see one of them in the homework.

Basically the whole idea of this query algorithm is that you keep — either you've already found a good estimate, or you know something about the next pivot. And then you get something better. And at the very end, you're guaranteed that you'll be able to return something, and if you're there you know this is a good approximation.

Student Question. *A question about the underlying model: What would happen if we allowed the query algorithm access to the underlying graph?*

Answer. Then you could do much more. The point here is to compress the graph so that we don't have to access it. If the query time is constant and you could look at the graph, you can't look at all the graph, but you could still do more — there's some work on compact routing that's related to this (you can have every node store some neighborhood that has different information from what other nodes store, and you can only read things in your local memory).

§15 April 3, 2025 — Algebraic algorithms for matching

We spent some time talking about how to avoid using MM in various scenarios related to shortest paths and cycles. Now we'll get back to using algebraic techniques. Today and next lecture, we'll discuss the maximum matching problem in general (unweighted, undirected) graphs.

§15.1 Perfect vs. maximum matchings

Definition 15.1. A *matching* in a graph G is a node-disjoint set of edges.

Problem 15.2 (Max Matching)
Find a matching of maximum size.

Problem 15.3 (Perfect Matching)
Find a matching of size $n/2$ (if one exists).

(This means a matching that has every node.)

Here's what's known about these problems. It's known that Max Matching and Perfect Matching are equivalent in terms of runtime. Clearly you can reduce Perfect Matching to Max Matching (you can find a maximum

matching and check whether its size is $n/2$). But what about the other direction? Say Perfect Matching is in $T(n, m)$ time (where n is the number of vertices, and m is the number of edges). How can I use an algorithm for Perfect Matching to solve Max Matching?

Here's a simpler question — what if instead of solving Max-Match, we want to solve the question: Given G and some integer k, does G have a matching of size at least k? Let's say I only have access to an algorithm for perfect matching, and I want to run it on some graph.

The idea is to add extra vertices. So we take our graph G, and we want to know if there's a matching of k edges. Then we have to do something with the $n - 2k$ remaining vertices. So what we'll do is add $n - 2k$ dummy vertices. And we put all possible edges betwen the dummy vertices and G.

Now you run your perfect matching algorithm on this new graph G'. If there were a matching of size k, then you had $2k$ vertices that were in that matching; the rest of the $n-2k$ nodes have a possibility to match with the dummies, so you can complete it to a matching with the dummies. So if there is a matching of size k, then there is a PM in G'. If there is a PM in G', then the dummy guys can match to at most $n-2k$ vertices in G (and they can't match to each other, so this is exact). Then you look at the vertices not matched to dummies; they must be matched to each other, so we get a matching on G of size k.

So the size of the graph at most doubles, and you can solve this problem. And then you can solve Max Matching by binary searching on k. So with $\log n$ iterations, you can solve Max Matching using an algorithm for Perfect Matching.

So these two problems are equivalent, at least in general graphs.

§15.2 What's known

What do we know for algorithms? Typically people care about bipartite graphs, and the bipartite graph case has been studied to death. There was a long line of work using flow techniques, and now we have an algorithm with runtime near-linear — $m^{1+o(1)}$. (This is from 2022, by a long list of authors — Chen, Kyng, Liu, Peng, Probst, Sachdeva.)

This really uses the fact that the graphs are bipartite. In general graphs, it's a different story. There's an $O(m\sqrt{n})$ time algorithm which was first claimed by Micali–Vazirani in 1980, and then one by Gabow–Tarjan 1981, and another by N. Blum 1990s. You might ask why the latter two; this is because Micali–Vazirani didn't provide a formal proof that the algorithm worked. They worked pretty hard for a long time, and maybe last year Vazirani wrote a general paper proving it. But then there were other algorithms that actually achieved this runtime.

This is a combinatorial algorithm, and the best one known for sparse graphs. But for dense graphs you can get an $\widetilde{O}(n^\omega)$ time algorithm, which is better when the graph is dense. This is due to Harvey, but it's based on an algorithm we'll talk about today by Rabin–Vazirani, which gives an n^ω time runtime to check whether a graph has a perfect matching (and then using the reduction from before, you can also find the cardinality of the maximum matching). But they were only able to get $\widetilde{O}(n^{\omega+1})$ time to find the matching (they could determine its size in n^ω, but could only find it in $n^{\omega+1}$; Harvey showed you don't need the $+1$).

Today we'll talk about the Rabin–Vazirani result, and next class we'll finish off with Harvey's.

This is the best known algorithm for dense graphs; there's no combinatorial algorithm that comes close. This is strange, because Virginia doesn't think you should need matrix multiplication to solve general graph matching; but that's where we are.

§15.3 The Tutte matrix

Here $G = (V, E)$ is some undirected unweighted graph. And we're going to define the *Tutte matrix*.

First we'll assume the vertices have names, and they're lexicographically sorted in some order — so there's some underlying order of V. And for every edge (u, v) with $u < v$, we create a variable x_{uv}. So we have m variables.

Definition 15.4. The *Tutte matrix* T is defined by

$$T(i,j) = \begin{cases} 0 & \text{if } i = j \text{ or } (i,j) \notin E \\ x_{ij} & \text{if } i < j \text{ and } (i,j) \in E \\ -x_{ji} & \text{if } i > j \text{ and } (i,j) \in E. \end{cases}$$

So this is some skew-symmetric matrix, which has 0's along the diagonal.

Why is this interesting? There's this theorem (which we will prove):

Theorem 15.5

We have $\det(T) \neq 0$ if and only if G has a perfect matching.

What is this matrix? This matrix has monomials as its entries, so the determinant is some polynomial of degree n. And this is saying that this polynomial is not the all-0's polynomial if and only if G has a perfect matching.

Remark 15.6. This is also true mod p for any prime p.

Proof. We're going to write down what the determinant is (we'll use two different formulas, but let's look at the first) — we have

$$\det(T) = \sum_{\sigma \in S_n} (-1)^{\text{sgn}(\sigma)} \cdot \prod_{i=1}^{n} T(i, \sigma(i)).$$

So we're summing over all permutations σ of n elements. You take all possible permutaitons; for every i you look at what the permutation takes it to, and you take the corresponding entry of the matrix, and you multiply them over all guys. The *sign* of a permutation σ has various interpretations. One is the number of inversions in the permutation — the number of $j < i$ such that i comes before j in the permutation. Another way to define it is as the parity of the number of even cycles in σ.

Example 15.7

If we have a permutation $(123)(45)(67)(9\,11)$, this means 1 is mapped to 2, 2 to 3, 3 to 1, 4 to 5, 5 to 4, and so on (we omit singletons, so 8 and 10 are mapped to themselves). This permutation has 3 even cycles, so its sign is 1 (because 3 is an odd number).

So this formula for the determinant looks over all permutations and sums all these terms. Let

$$(-1)^{\text{sgn}(\sigma)} \cdot \prod_{i=1}^{n} T(i, \sigma(i)) = T_\sigma.$$

Now, the first part of our proof will be, let's suppose the graph does have a perfect matching. Then we will show that the determinant is not the zero polynomial. Suppose the perfect matching is $M = \{(i_1, i_2), (i_3, i_4), \ldots, (i_{n-1}, i_n)\}$ (where each of these is an edge; notably n must be even). And let's say without loss of generality that we've sorted the vertices according to the matching (otherwise there may be some $-$ signs), so $i_1 < i_2 < \cdots < i_n$.

Now we'll consider the following permutation σ_M, which is just the product of these edges viewed as cycles in the permutation — so we take

$$\sigma_M = (i_1 i_2)(i_3 i_4)\cdots(i_{n-1} i_n)$$

(this means $i_1 \mapsto i_2$, $i_2 \mapsto i_1$, and so on). Its sign is $n/2 \pmod 2$, because there's $n/2$ cycles (all of which are even). Let's consisder T_{σ_M}; this is

$$T_{\sigma_M} = (-1)^{n/2} \cdot T(i_1, i_2)T(i_2, i_1)\cdots T(i_{n-1}, i_n)T(i_n, i_{n-1}).$$

(This is because $i_1 \mapsto i_2$, $i_2 \mapsto i_1$, and so on.) Now because of the way this is defined, we have $T(i_1, i_2) = -T(i_2, i_1)$. So what we get is

$$(-1)^{n/2} x_{i_1 i_2}^2 x_{i_3 i_4}^2 \cdots x_{i_{n-1} i_n}^2 \cdot (-1)^{n/2},$$

where the $(-1)^{n/2}$ at the end is because you have a -1 from each term. And the two $(-1)^{n/2}$ factors cancel, so you get

$$T_{\sigma_M} = x_{i_1 i_2}^2 \cdots x_{i_{n-1} i_n}^2.$$

Now we claim there's no other term which will get this term (even with a coefficient). That's because there's no way to get $x_{i_1 i_2}^2$ unless i_1 and i_2 are their own cycle.

So this term will not be cancelled (even mod p) — no other T_σ can cancel this term. So $\det(T)$ is not the zero polynomial, even mod p. (Basically this term survives.)

Student Question. *What was the WLOG $i_1 < i_2 < \cdots$ assumption for?*

Answer. Just because we want to write $x_{i_1 i_2}$ — if we didn't know $i_1 < i_2$, we wouldn't know which order to write. (It's just for clarity.)

Student Question. *Is p a specific prime, or is it anything?*

Answer. It's any prime — the point is that the coefficient is 1, which is nonzero mod every prime.

Now suppose that $\det(T) \neq 0$. What we're going to do is we'll consider these terms T_σ; and we're going to split permutations into two types. One type is permutations that have an odd cycle (ignoring singletons); and the other is ones that don't. And we'll pair up permutations that have an odd cycle, and we'll show that all the terms corresponding to permutations that have odd cycles will just cancel. And so $\det(T)$ will actually be a sum over permutations that only have even cycles.

Now, we want to point out that we will not care about permutations that have fixed points. Why? Our matrix T has 0's on the entries (i, i) for every i, so we won't be caring about any permutations that have fixed points (cycles of length 1) — they have terms of 0. So we only care about permutations that have an odd cycle of length at least 3.

Now, consisder all permutations σ that have an odd cycle C_σ (which has length at least 3). Now what we'll do is for every such permutation, we'll find a unique permutation σ' that we'll match to σ; and this will form a bijection.

In order to do that, we're going to pick a particular odd cycle — if the permutation has multiple odd cycles, we take C_σ to be the odd cycle that has the minimum element out of all the odd cycles in the permutation.

Example 15.8

For example, if $\sigma = (12)(356)(748)$, then we have two odd cycles, and we pick (356) to be C_σ, because it has the smallest element (3 vs. 4).

This uniquely picks out a particular odd cycle in every permutation that has one.

Then given this, let σ' be the permutation with the same cycles as σ, except that C_σ is reversed.

Example 15.9

So for example, for this pmerutation we'd have $\sigma' = (12)(653)(748)$ — the first and last cycles are the same as before, but the middle one is reversed.

Now, this sort of matching between σ and σ' forms a bijection between the permutations with odd cycles — if σ is mapped to σ', then σ' is mapped to σ (they have the same other cycles, and for the unique odd cycle with the smallest element, we've reversed it).

Now, let's consider — I'm going to consider $T_\sigma + T_{\sigma'}$. Notably, σ and σ' have exactly the same number of even cycles (their cycle structure is the same; they only differ in a single odd cycle having been reversed), so their signs are the same — $\operatorname{sgn}(\sigma) = \operatorname{sgn}(\sigma')$, so we can factor out $(-1)^{\operatorname{sgn}(\sigma)}$. Then we have products of $T(i, \sigma(i))$. These terms will be the same for every i that's not in our particular odd cycle C_σ, so we can just factor them out; and we'll get

$$(-1)^{\operatorname{sgn}(\sigma)} \prod_{i \notin C_\sigma} T(i, \sigma(i)) \cdot \text{something}.$$

And what is that something? These are the terms that correspond to C_σ — so this is

$$\prod_{i \in C(\sigma)} T(i, \sigma(i)) + \prod_{i \in C(\sigma)} T(i, \sigma'(i)).$$

But σ' is the reversal of σ in this particular cycle, so

$$\prod_{i \in C_\sigma} T(i, \sigma(i)) = \prod_{i \in C_\sigma} T(\sigma(i), i) = \prod_{i \in C_\sigma} -T(i, \sigma(i)).$$

So this second term is $(-1)^{|C_\sigma|}$ times the first term. And $(-1)^{|C_\sigma|} = -1$, because C_σ is an odd cycle. So we get

$$(-1)^{\operatorname{sgn}(\sigma)} \prod_{i \notin C_\sigma} T(i, \sigma(i)) \cdot \left(\prod_{i \in C_\sigma} T(i, \sigma(i)) + (-1) \prod_{i \in C(\sigma)} T(i, \sigma(i)) \right) = 0.$$

And this means all the terms corresponding to permutations that have an odd cycle just cancel. So actually, $\det T = \sum_\sigma T_\sigma$, but where we sum only over the permutations that only have even cycles.

Why do we care about permutations that only have even cycles? The point is for any permutation that only has even cycles, you can extract a matching out of it. Suppose I have some permutation σ with only even cycles, and T_σ is nonzero. Then consider any one of the even cycles; say it's $C = (i_1 i_2 \cdots i_{2k})$. Becaues the determinant just has the product of the terms, since $\prod_{i \in C} T(i, \sigma(i)) \neq 0$, we get that (i_1, i_2), (i_3, i_4), $\ldots$, (i_{2k-1}, i_{2k}) are all edges in E. So you take any even cycle in your pmerutation and alternate (taking $i_1 i_2$, $i_3 i_4$, and so on); all of those are edges because this term is nonzero. So when you take this collection of edges over all even cycles, they form a perfect matching.

Example 15.10

If $\sigma = (1\,10)(2468)(35)(79)$, then your matching will be the edges $(1, 10)$, $(2, 4)$, $(6, 8)$, $(3, 5)$, $(7, 9)$.

So as long as your determinant is not the zero polynomial, you will be able to extract one of these edges, and you'll get your perfect matching. □

§15.4 Evaluating the Tutte matrix

Given this, what's one way to get an n^ω time algorithm to check whether the graph has a PM? Well, we could construct the matrix and compute its determinant and check if it's 0. Unfortunately, this matrix has

variables in it, so we'd get a huge polynomial, which means computing that polynomial is not so easy. But we *could* just evaluate it on some random numbers — we could plug random numbers into it.

We'll use the following theorem or lemma (which is due to many people):

Lemma 15.11 (Schwarz–Zippel, De Millo–Lipton, ...)

Suppose P is a degree d polynomial, and suppose we pick random values (denoted by a vector v) for the variables from some $S \subseteq \mathbb{F}$. Then if P is not the zero polynomial, we have

$$\mathbb{P}_v[P(v) \neq 0] \geq 1 - \frac{d}{|S|}.$$

So I pick some large set in my underlying field to evaluate the polynomial on, and pick random values for all the variables; and I get that the probability it stays nonzero is $1 - d/|S|$. So I can get rid of the variables by plugging in random values, and I can still probabilistically check whether the polynomial is the zero polynomial or not. (If the polynomial is zero then you'll always get 0; if it's not, then you'll be able to catch it as long as the subset of the field is large enough.)

In our case we have $\deg(\det T) = n$ (it's the determinant), so it suffices to pick $|S| \sim n^2$ (for example).

The field we evaluate things over actually matters — we can't just pick a subset of n^2 integers. The problem is that the determinant has degree n, so if you just evaluate over $\mathbb{R}$, the entries blow up (they become exponential in the sizes that you plug in). And when evaluating a determinant, we have to take into account the sizes of the entries in the matrix.

But we did this dance where we said we could also pick any prime we want. So if we pick our field to be $\mathbb{F} = \mathbb{Z}_p$ for some prime p of size roughly n^2, then when we evaluate the polynomial, we can always do things mod p; and the integers you care about will always be $O(\log n)$ bits. SO our computation of the determinant won't have to deal with large numbers, which means we can evaluate it mod p.

So as a recap, you take your graph and form the Tutte matrix. Then you take some big prime p of size roughly n^2. Then we take S to be the whole field $\mathbb{Z}_p$, we pick random values for the variables x_{ij}, and substitute. Then we check whether the determinant is 0 or not. Computing the determinant only takes $O(n^\omega)$ time; and we'll know whether the determinant is nonzero with probability at least $1 - 1/n$. (If we pick a prime that's biugger, then you boost the probability; you can also repeat.)

So now we know how to check for a perfect matching in time n^ω with a randomized algorithm; this is the first part of Rabin–Vazirani.

§15.5 Finding a perfect matching

Now, how can we use an algorithm that can *check* if a graph has a PM to also *find* a PM?

One way to do this is to take 1, remove half its edges and see if you still have a PM, and then recurse (and do this for every vertex); this should give you $\log n$ overhead. But we'll see a different algorithm that has ideas leading to Harvey.

Right now, we'll see a different way to have an overhead of n; next time we'll try to do it more cleverly to not have the overhead.

From now on, we'll imagine that we've already evaluated T on random points — so T is now no longer a matrix containing variables, but a matrix over $\mathbb{Z}_p$.

Here's what we do. Before, we saw one formula for $\det(T)$. But now we'll see a different one, which is maybe the first formula you learn in high school — we have

$$\det(T) = \sum_{j=1}^{n} (-1)^{1+j} T(1,j) \cdot \det(T_{\{1\},\{j\}}).$$

What is this notation? Here, for two subsets $X, Y \subseteq [n]$, we write $T_{X,Y}$ to denote T with all rows in X and columns in Y removed. So you take the top-left entry and multiply by the determinant of the bottom-right rectangle, then the next entry, and so on; and you sum all these up (with signs).

And we've evaluated the Tutte matrix and found that its determinant is nonzero, so a PM exists; and we now want to *find* this perfect matching. So if $\det T \neq 0$, that means one of these terms is nonzero — that means there exists j such that $T(1,j) \neq 0$ and $\det(T_{\{1\},\{j\}}) \neq 0$. The first part means that the edge $(1,j)$ exists. And we kind of want $\det(T_{\{1\},\{j\}}) \neq 0$ to mean that the graph when you remove 1 and j still has a perfect matching. This turns out to be true (we'll prove it later).

Claim 15.12 — If $\det(T_{\{1\},\{j\}}) \neq 0$, then $\det(T_{\{1,j\},\{1,j\}})$ is also nonzero.

And the second part means that $G \setminus \{1,j\}$ (the graph without the vertices 1 and j) has a perfect matching.

So if I could find some j hwere both terms are nonzero (we know it must exist), then I could just put the edge $(1,j)$ in the perfect matching, and recurse on the graph where I've removed two vertices.

So we want to find this j. Now we're going to use the fact that a long time ago, we showed that we can compute the inverse of a matrix in the same time as matrix multiplication. So now we're going to look at T^{-1}. We know $\det T \neq 0$, so it's invertible, and we can look at its inverse. And actually, the (i,j)th entry of the inverse has a very nice formula — something like

$$T^{-1}(i,j) = (-1)^{i+j} \cdot \frac{\det(T_{\{i\},\{j\}})}{\det(T)}.$$

(This is the adjoint formula, or something.)

Suppose I can compute T^{-1}; now how do I find the $(1,j)$ entry that has the two properties we want? Suppose I take my Tutte matrix and compute its inverse; what does this formula give me about it? We have $T^{-1}(i,j) \neq 0$ if anad only if $\det(T_{\{i\},\{j\}}) \neq 0$.

So here's an algorithm that gives what I want:

Algorithm 15.13 (Perfect matching)

Suppose we have a random evaluation T of the Tutte matrix.

- If $\det T = 0$, return NO.
- Otherwise, initialize $M \leftarrow \emptyset$.
- Compute $N \leftarrow T^{-1}$, and find j such that $(1,j) \in E$ and $N(1,j) \neq 0$.
- Recurse on $T_{\{1,j\},\{1,j\}}$ to get some matching M', and return $M = M' \cup \{(i,j)\}$.

We spend n^ω time computing $\det(T)$ and T^{-1}. Then finding j takes you n time. And then you have to repeat this $n/2$ times, so the runtime is $\widetilde{O}(n \cdot n^\omega) = \widetilde{O}(n^{\omega+1})$. And with high probability you'll be able to find a PM.

What we're going to do next time — we're not actually done because we haven't proven Claim 15.12, but we'll do it in a little bit — is we'll see that when we remove $\{1,j\}$ from the Tutte matrix, it doesn't change very much. So there's no real reason to keep recomputing the inverse (which is the expensive part). Next time, we'll try to find some ways to *dynamically update* our inverse, so we don't have to keep recomputing it. This will lead to an n^ω time algorithm, as opposed to $n^{\omega+1}$.

Remark 15.14. You aren't really recursing. Basically you can write it as, until the matching is of size $n/2$, you keep updating the matrix and adding edges to the matching.

§15.6 Proof of Claim 15.12

What we're missing is that the formula tells us there's one term where $(1, j)$ is an edge, and the determinant when I remove row 1 and column j is nonzero. But to guarantee that the graph when we remove vertices 1 and j has a PM, we actually need that the determinant when we remove rows and columns 1 and j is nonzero. So what we want to prove is that if $\det(T_{\{i\},\{j\}}) \neq 0$, then $\det(T_{\{i,j\},\{i,j\}})$ is also nonzero; this means that when we recurse, we'll be able to find the perfect matching.

To prove this claim, we'll use some lemma that we're not going to actually prove, but it's very useful.

Lemma 15.15

Let A be any $n \times n$ skew-symmetric matrix. Then the following hold:

(1) A^{-1} is skew-symmetric.

(2) If n is odd, then $\det(A) = 0$.

(3) Let $Y \subseteq [n]$ be any subset of columns such that $|Y| = \text{rank}(A)$, and the column rank of $A(\bullet, Y)$ is $\text{rank}(A)$. Then $\det(A(Y, Y)) \neq 0$.

In our case, (2) is obvious — we know $\det(T) = 0$ if and only if there is no PM, and of course if the graph has an odd number of vertices then it can't have a PM, so the determinant is 0. But it's also true for any $n \times n$ skew-symmetric matrix when n is odd.

The third property is the most important, and the one we're actually going to use. The picture is that we have A, and let's say it doesn't have full rank; and there's some subset of columns which form Y. If we restrict A to these columns, then we want these columns to be linearly independent, and their number should be the rank of A. And then — we've picked the columns in Y, but if we pick the corresponding *rows* as well, then the corresponding square $Y \times Y$ matrix is supposed to be full rank (and this rank is $\text{rank}(A)$). We're not going to prove it; you can try proving it by yourself if you want. But what it says is if A is skew-symmetric and its rank is not full, and I can find some columns which are independent and account for the rank, then if I take those rows and columns together, I get a full-rank matrix (whose rank is the same as the original).

Now we're going to use this property to prove Claim 15.12.

Proof of Claim 15.12. For simplicity, assume without loss of generality that $j = 2$, and assume that $\det(T_{\{1\},\{2\}}) \neq 0$. So we have our matrix T, and if I remove the first row and the second column, then the remaining matrix has a nonzero determinant and full rank. And we want to prove that if I remove rows and columns 1 and 2, I will also have a full rank matrix.

The first thing we'll do is, suppose I consider the matrix $T_{\{1\},\{1\}}$ — so I take my submatrix where I remove row and column 1. This is still a skew-symmetric matrix — it's the Tutte matrix of the graph where I removed row and column 1. And n was even, so when I remove 1, I get a skew-symmetric matrix of odd dimension. And therefore $\det(T_{\{1\},\{1\}}) = 0$.

Now, this is a $n - 1$ by $n - 1$ matrix, and if it doesn't have full rank, that means $\text{rank}(T_{\{1\},\{1\}}) \leq n - 2$.

Now, we have this assumption that $\det(T_{\{1\},\{2\}}) \neq 0$. This assumption means when I remove row 1 and column 2, I get a full-rank matrix — so the rank of this thing is $\text{rank}(T_{\{1\},\{2\}}) = n - 1$.

That means if I remove columns 1 and 2, the column rank will become $n - 2$. So if I consider $T_{\{1\},\{1,2\}}$ (removing both columns 1 and 2), the column rank is $n - 2$ — the original had $n - 1$ independent columns, and when I remove one of them (namely 1) I remove one extra column; they're still linearly independent, but now I have two of them.

And now $T_{\{1\},\{1,2\}}$ is a submatrix of $T_{\{1\},\{1\}}$ where I took this and removed column 2. We saw that the latter has rank at most $n-2$. But it actually has $n-2$ independent columns (namely, all the columns other than 2). So it must have rank *exactly* $n-2$.

Now we're going to use the third property. Now we have that $T_{\{1\},\{1\}}$ is $(n-1)\times(n-1)$ and has rank $n-2$. So we have our picture with T, and $T_{\{1\},\{1\}}$ means I removed row and column 1 (so now I get a box in the bottom-right). And now consider Y (for the lemma) consisting of all the columns 3, 4, ..., n (so excluding 1 and 2). Then $|Y| = n-2$. Now, what do we need for that lemma? We need that the submatrix of $T_{\{1\},\{1\}}$ restricted to the columns in Y has full rank. But what is this? If we restrict $T_{\{1\},\{1\}}$ to Y, this is exactly $T_{\{1\},\{1,2\}}$ (where we remove column 2 as well). And we said earlier that that thing has column rank $n-2$. This is exactly what we need for the lemma.

So by the lemma applied to Y, what we get is that $\det(T_{\{1\},\{1\}}(Y,Y)) \neq 0$. But what is this matrix? Well, Y is from 3 to n, and we're doing this for both rows and columns. So I get the square in the bottom-right, which is exactly $T_{\{1,2\},\{1,2\}}$.

So when I restrict T (with rows 1 and 2 and columns removed), it's exactly just T with both the rows and columns 1 and 2 removed. From this, we get that

$$\det(T_{\{1,2\},\{1,2\}}) \neq 0. \qquad \square$$

That kind of finishes the proof; and as long as we can find some T where row 1 and column j removed causes nonzero determinant, we've actually found a candidate edge to put in our matching.

This finishes that algorithm; next time we'll see how to update the inverse efficiently, so that we don't have to pay the n overhead.

§16 April 8, 2025 — Perfect Matchings, Part 2

Today we're going to continue our presentation of the fastest-known algorithm for perfect matching in general graphs.

§16.1 Review

Last time, we looked at the Tutte matrix of a graph, which is a matrix made of variables x_{ij}:

Definition 16.1. The *Tutte matrix* is defined by

$$T(i,j) = \begin{cases} 0 & \text{if } i = j \text{ or } (i,j) \notin E \\ x_{ij} & \text{if } i < j \text{ and } (i,j) \in E \\ -x_{ji} & \text{otherwise.} \end{cases}$$

Lemma 16.2

We have $\det(T) \neq 0$ if and only if G has a perfect matching.

And then we said that if we plug in random values to T over $\mathbb{F}_p$ for $p \geq \Omega(n^2)$, then we can compute this random evaluation of the determinant fast (in n^ω time). And if there's no perfect matching then of course we'll still have $\det T = 0$; and with high probability, if G has a perfect matching, then $\det T$ remains nonzero.

This gives a randomized $\widetilde{O}(n^\omega)$ *detection* algorithm for perfect matchings.

From now on, we'll work with this random evaluation of the Tutte matrix. So now we can check with high probability whether a graph has a perfect matching; but we'd also like to *find* one. We gave an algorithm which runs in $\widetilde{O}(n^{\omega+1})$ time:

Algorithm 16.3 (Rabin–Vazirani)

Let T be a random substitution of the Tutte matrix. If $\det(T) = 0$, return No.

Initialize $M \leftarrow \emptyset$.

While $|M| < n/2$:

- Compute $N \leftarrow T^{-1}$.
- Find j such that $N(1,j) \neq 0$ and $T(1,j) \neq 0$.
- Set $M \leftarrow M \cup \{(i,j)\}$ and $T \leftarrow T_{\{1,j\},\{1,j\}}$.

Return M.

So we start trying to build up the matching edge by edge. While we haven't yet computed a full matching, we recompute the determinant of the Tutte matrix. Then we find some j — this is an edge $(1,j)$ (because $T(1,j) \neq 0$). And last time, we showed that if $T^{-1}(1,j) \neq 0$, then actually removing 1 and j from the graph will leave us a graph with two fewer vertices that also has a perfect matching — so this condition says that the edge $(1,j)$ appears in some perfect matching.

So once we have these two things, we know $(1,j)$ can be put into the perfect matching. So we add it to the perfect matching, and modify our Tutte matrix by removing the rows and columns corresponding to 1 and j (and then we recompute the inverse, and so on).

This entire thing involves computing $n/2$ inverses, so it'll run in $n^{\omega+1}$ time.

Today we'll do two things. The first is the Mucha–Sankowski algorithm, whose point is that we don't actually need to recompute T^{-1} every time — you can actually update it in n^2 time at each step. So M–S can update n faster, and that'll lead to an $O(n^3)$ algorithm. In the end, we'll also give an $\widetilde{O}(n^\omega)$ algorithm (due to Harvey). But that's not based on this paradigm (of finding an edge in some perfect matching) — that's based on finding edges that can be removed such that there is still a perfect matching. We'll keep finding such edges, and it'll actually be very cheap to do so; and what you'll end up with in the end is the edges of some perfect matching.

§16.2 Mucha–Sankowski — faster inverse updates

Imagine you have your random evaluation of the Tutte matrix T, and you've already computed the inverse once — so you have T, and you also have $N = T^{-1}$. And now you've found some edge $(1,2)$ (we previously had $(1,j)$, but we can reorder to assume $j = 2$) such that $T(1,2) \neq 0$ and $N(1,2) \neq 0$. And then we want $T_{\{1,2\},\{1,2\}}$; and we want the inverse of this.

Let's draw this, since when we draw things it's nicer.

We'll name these pieces as

$$T = \begin{bmatrix} X & Z \\ Y & W \end{bmatrix} \quad \text{and} \quad N = \begin{bmatrix} \hat{X} & \hat{Z} \\ \hat{Y} & \hat{W} \end{bmatrix},$$

and we're interested in W^{-1}.

First let's look at $\widehat{X}$. We know the inverse of a skew-symmetric matrix is skew-symmetric, so we'll have 0's on the diagonal; and we'll have

$$\widehat{X} = \begin{bmatrix} 0 & N(1,2) \\ -N(1,2) & 0 \end{bmatrix}.$$

We picked $(1,2)$ such that $N(1,2) \neq 0$; so both these off-diagoal entries are nonzero. That means $\widehat{X}$ is invertible — we have $\det(\widehat{X}) = N(1,2)^2 \neq 0$. And it's a 2×2 matrix, so we can compute its inverse in constant time — we have

$$(\widehat{X})^{-1} = \begin{bmatrix} 0 & -1/N(1,2) \\ 1/N(1,2) & 0 \end{bmatrix}$$

or something like that. This is an $O(1)$ time computation, to get this inverse.

Now, what we're going to do is we're going to actually give a formula for W^{-1}, given that I know $\widehat{X}$ is invertible and I have its inverse.

So we're going to look at — there's this portion YW, and we're going to multiply it by the parts of N and see what we get. First let's consider

$$\begin{bmatrix} Y & W \end{bmatrix} \begin{bmatrix} \widehat{X} \\ \widehat{Y} \end{bmatrix}.$$

This is supposed to be 0, because N is an inverse of T (and this part of the matrix product corresponds to some block of TN which doesn't touch the diagonal, so it's all-zeros). And what is it? It's $Y\widehat{X} + W\widehat{Y}$. So we get

$$0 = Y\widehat{X} + W\widehat{Y}.$$

We can move things around to get $Y\widehat{X} = -W\widehat{Y}$. And now we can use the fact that $\widehat{X}$ is invertible, so we get

$$\boxed{Y = -W\widehat{Y}(\widehat{X})^{-1}}.$$

Now I'll consider

$$\begin{bmatrix} Y & W \end{bmatrix} \begin{bmatrix} \widehat{Z} \\ \widehat{W} \end{bmatrix}.$$

Now, this is supposed to be the $(n-2) \times (n-2)$ identity matrix I. So from this, I get

$$I = Y\widehat{Z} + W\widehat{W}.$$

Now I plug in what Y is from before, and I get

$$-W\widehat{Y}(\widehat{X})^{-1}\widehat{Z} + W\widehat{W} = I.$$

Now some nice stuff happens, because W appears twice. So I get

$$W(\widehat{W} - \widehat{Y}\widehat{X}^{-1}\widehat{Z}) = I.$$

That means this remaining part is the inverse of W — we get

$$W^{-1} = \widehat{W} - \widehat{Y}(\widehat{X})^{-1}\widehat{Z}.$$

That's fine — this is just some formula — but let's look at the *dimensions* of the matrices in this fomula. The first term $\widehat{W}$ is just $(n-2) \times (n-2)$. For $\widehat{Y}$, it's $(n-2) \times 2$. And $\widehat{X}^{-1}$ is 2×2, and $\widehat{Z}$ is $2 \times (n-2)$. Basically, what's going on here is you're multiplying a roughly $n \times 2$ matrix (this very skinny guy) by a tiny thing, and then by another skinny thing in the other direction. And computing the first part ($(n-2) \times 2$ by 2×2) only takes $O(n)$ time. From this you get an $(n-2) \times 2$ matrix. And now you have to multiply

it by this $2 \times (n-2)$ matrix. And this takes n^2 time — just by the brute-force algorithm (which does the product of the dimensions).

So this part $\widehat{Y}(\widehat{X})^{-1}\widehat{Z}$ can be computed in $O(n^2)$ time. And then all you have to do is subtract it from a matrix with n^2 entries. This means the full recomputation of this matrix W^{-1} can be done in $O(n^2)$ time, via brute force! (You don't need fancy matrix multiplication machinery for this.)

In fact, what you can do is in the very initial phase when you compute $N \leftarrow T^{-1}$, you can just do it with Gaussian elimination (which would be $O(n^3)$). Then every time you recompute, you take $O(n^2)$ time; and you do this $n/2$ times, so your total runtime is $O(n^3)$.

So we immediately save from $O(n^{\omega+1})$ to $O(n^3)$, without anything fancy.

Student Question. *Could you do slightly better by not recomputing the entire thing, just the parts you need?*

Answer. Maybe. We know this problem can be solved faster combinatorially, so there should be some way. We'll see how to do something different that's kind of similar, but Virginia doesn't know how to do better than n^3 with this paradigm; but maybe it's possible, and that would be nice.

§16.3 Harvey's algorithm

Now we're going to move on to Harvey's algorithm, which is somewhat more involved than this. But it's based on the following naive algorithm.

Algorithm 16.4 (Naive-Matching(G))

For every edge $e \in E$:

- If $G \setminus e$ has a perfect matching, then remove e from G.

Return E.

So for every edge, we check, can I remove it so that there's still a perfect matching? If so I just remove it, and now I have the graph without one edge; and I try again. I do this n^2 times, and in the end I just return the remaining edges.

This is an extremely silly algorithm. We know we can check whether $G \setminus e$ has a perfect matching in $\widetilde{O}(n^\omega)$ time (using the Tutte matrix). So you get $n^{\omega+2}$, which is completely useless. But you can still use the *idea* of this algorithm to compute things much faster.

The idea of how we'll improve on this is that in order to check whether $G \setminus e$ has a perfect matching, you don't have to know the full inverse of the Tutte matrix — in fact, you just need a 2×2 *submatrix* of the inverse. This will let us get a recursive algorithm that tries to remove edges from some subset of the graph; and once it figures out which edges can be removed, it removes them and only updates the appropriate submatrix of the inverse (so you don't have to update too many things). That's the rough idea; but we'll have to prove some things and give some lemmas.

We'll start with a main lemma: Suppose we have our Tutte matrix, and we already have the inverse of the Tutte matrix. And now we want to see what happens if I change the Tutte matrix by setting a few entries to 0, but they're localized in a small $s \times s$ square of the matrix. Then when I set these entries to 0, I want to know whether the determinant becomes 0 or not. The point of this lemma is that in order to check that, it'll only take me s^ω time.

Lemma 16.5

Suppose we're given an $n \times n$ skew-symmetric matrix M and $N = M^{-1}$, and we have some subset $S \subseteq [n]$ (of the rows and columns) and some $\widetilde{M}$, which is the same as M except possibly on some entries in $M[S, S]$ — i.e.,

$$\widetilde{M} = M + \Delta,$$

where Δ is all-0's except in $\Delta[S, S]$.

Then let Γ be the $|S| \times |S|$ matrix defined by

$$\Gamma = I_{|S|} + \Delta(S, S) \cdot M^{-1}[S, S].$$

(1) $\widetilde{M}$ is invertible if and only if $\det(\Gamma) \neq 0$.

(2) If $\widetilde{M}$ is invertible, then $\widetilde{M}^{-1} = M^{-1} - M^{-1}[\bullet, S]\Gamma^{-1}\Delta[S, S]M^{-1}[S, \bullet]$.

Think of M as the original Tutte matrix, and $\widetilde{M}$ as what happens if I decide to remove some edges in $S \times S$ — I want to try removing them and see whether the determinant of the Tutte matrix becomes 0.

We're not going to prove this, but you can — you can do it by multiplying the right-hand side by $\widetilde{M} + \Delta$, and you'll see things will cancel out and you'll get the identity.

Student Question. *For the first part, it's easy to see going left, but how would you prove it going right?*

Answer. In a similar way — you can get the inverse of Γ from the inverse of $\widetilde{M}$.

But we'll give some consequences of this.

Student Question. *When you write $M^{-1}[\bullet, S]$, that means you take all the rows?*

Answer. Yes — you take all the rows, and all the columns indexed by S.

In our application, we want to check whether removing some edges in $S \times S$ will leave us with a graph that still has a perfect matching. And from this lemma, we see that the only thing we need to do is compute the determinant of an $|S| \times |S|$ matrix. So if $|S| \ll n$, then we actually need to do much less than n^ω computation to check, which is great. In particular, if I want to check whether a single edge can be removed, this can be done in *constant* time!

The second part is, suppose I happen to figure out that some subset of edges I've picked, I can just remove them. Then I can actually update this matrix. Unfortunately the way it's written is kind of annoying (you'd have to update it in some matrix multiplication time depending on n by $|S|$ and so on). But it turns out if you only want to update an $|S| \times |S|$ submatrix of the inverse, you can do that in $|S|^\omega$ time. We'll design an algorithm which checks if you can remove some subset of edges like this, and if we can we only need to update an $|S| \times |S|$ submatrix.

Let's look at the special case where S is just two vertices. Suppose I have $S = \{i, j\}$ (where $i < j$). And I want to check if $G \setminus \{i, j\}$ has a perfect matching.

From this, I get to compute Γ for this set; we have

$$\Gamma = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} + \begin{bmatrix} 0 & -T(i,j) \\ T(i,j) & 0 \end{bmatrix} \cdot \begin{bmatrix} 0 & N(i,j) \\ -N(i,j) & 0 \end{bmatrix}$$

(in the second term, the first factor is $\Delta(S, S) = \widetilde{M}(S, S) - T(S, S)$ — the entries of the Tutte matrix that I need to cancel out in order to kill off the edge (i, j), since $\widetilde{M}(S, S)$ is all-0's; for the second factor, it's just

the inverse, and we defined $T^{-1} = N$; and this is a skew-symmetric matrix). So this is

$$\begin{bmatrix} 1 + T(i,j)N(i,j) & 0 \\ 0 & 1 + T(i,j)N(i,j) \end{bmatrix}.$$

So this is Γ; when is it invertible? It's invertible when

$$T(i,j)N(i,j) \neq -1.$$

So in order for me to know whether I can remove edge (i,j), I just need to know whether this is true. Notably, I only need to know a *single* entry of the inverse $N = T^{-1}$.

So this is extremely local, and it suggests some sort of algorithm where you take your graph, and first you take S to be the full vertex set. Then you split it into two halves. And then you recursively remove edges in the first half and the second such that there's still a perfect matching, and then try to remove edges between the two halves so there's still a perfect matching. Every time you do that, you only need to update the $S \times S$ submatrix of the inverse, because we only need the entries inside that submatrix.

So let's imagine I have some subset $S \subseteq V$, and let's say $\widehat{N}$ is our current estimate of N (the true inverse of the current (evaluation of the) Tutte matrix T). Imagine that we promise that in the entries $S \times S$, this estimate is actually correct — so

$$\widehat{N}[S,S] = N[S,S]$$

(we don't care about the other entries).

Let's assume all the sets we work with have sizes which are powers of 2 (to ignore floors and ceilings). Then I can take S and cut it into two parts S_1 and S_2 of size $\frac{1}{2}|S|$.

If I do this, I know $\widehat{N}[S,S]$ is correct, so in particular the $S_1 \times S_1$ submatrix is also correct. So I recurse in here. And what I get is a subset of edges inside $S_1 \times S_1$ that I can remove such that the graph still has a perfect matching. So we get some set of edges $U_1 \subseteq S_1 \times S_1$ such that $G \setminus U_1$ has a perfect matching.

Now I want to remove these edges. How do I remove them? I update the inverse — because once I recurse, I may have messed up the inverse by removing these edges. So I'll update the inverse $\widehat{N}$. But when I update, I don't update on just the $S_1 \times S_1$ part, but the entire $S \times S$ part — because I'll then recurse on the $S_2 \times S_2$ part, so I'll need the inverse to be correct here.

In other words, deleting U_1 will mess up the estimate $\widehat{N}(S,S)$, and we'll need it to be correct for the next part. So we fix it! How do we fix it? We'll define a Update procedure, which will be given T, the old $\widehat{N}$, and the new ste of edges U_1 that we're removing. And how do we update? Well, we follow the formula from Lemma 16.5 — Δ is defined by the edges that were removed. We look at Γ, which is defined by Δ (the edges we removed) and $M^{-1} = \widehat{N}$, which we know is correct on this $S \times S$ matrix.

To write this explicitly, $\mathsf{Update}(\widehat{N}, U_1)$ works as follows: First, we compute $\Delta(S,S)$ (which means we just zero out all the variables that have edges in U_1). Then after this, we compute

$$\Gamma = I + \Delta[S,S] \cdot M^{-1}[S,S].$$

We also compute Γ^{-1}. We know that $\widehat{N}(S,S)$ is correct (i.e., it equals $N(S,S)$ by assumption). Then we compute the inverse, but only on $S \times S$ — so we compute

$$\widehat{N}_{\text{new}}[S,S] \leftarrow \widehat{N}[S,S] - \widehat{N}[S,S]\Gamma^{-1}\Delta[S,S]\widehat{N}[S,S].$$

Student Question. *Hree when we say S, is it S_1 or S?*

Answer. It's S — we need to update for the whole S.

The difference between this and what's in Lemma 16.5 is that the $\bullet$'s became S's, because I only want a $S \times S$ submatrix.

What this is doing is it gives us an $O(|S|^\omega)$ time algorithm to fix the inverse in the $S \times S$ submatrix. This means we can fix the inverse in $\widetilde{O}(|S|^\omega)$ time.

So far what we've done is taken S with the promise that the inverse we have is correct in $S \times S$. We've recursively figured out what edges can be removed in S_1, and we try to remove them by updating the inverse. We updated in the full $S \times S$ matrix so that now we can next recurse in S_2. This will give us some new subset $U_2 \subseteq S_2 \times S_2$. Deleting U_2 can mess up the inverse, so we update it again (spending $|S|^\omega$ time).

Now the only edges we haven't checked whether we can remove are those between S_1 and S_2. For those, we'll give a new algorithm, which just tries to remove edges in $S_1 \times S_2$. It'll do bascally the same thing — it'll give some new set of edges $U_3 \subseteq S_1 \times S_2$. We remove them, and then we update.

We'll write down the whole algorithm for this, and then we'll analyze the whole runtime and see what's going on (and also the correctness).

We are making the naive matching algoritihm very very fast, so we'll erase it.

There will be two procedures. One procedure is the overlay procedure, which is doing this thing where we try to remove edges with endpoints in $S \times S$. It'll call a procedure that removes edges in $S_1 \times S_2$ assuming that S_1 and S_2 are disjoint.

First we'll define $\mathsf{DeleteCross}$. Suppose we have input (S_1, S_2), where $|S_1| = |S_2|$ and $S_1 \cap S_2 \neq \emptyset$ and $N(S, S)$ is known, where $S = S_1 \cup S_2$:

- **Base case:** If $|S_1| = |S_2| = 1$, then we're in the case where we're checking whether we can remove a single edge. So we're in the case $S_1 = \{i\}$ and $S_2 = \{j\}$, and we can just check the condition we saw before — if $T(i,j)N(i,j) \neq -1$, then we can remove the edge. (By our argument from before, we only need to know the entries in that particular position (i,j), and we assumed we know it.)
- Otherwise, chop S_1 and S_2 into (roughly equal) parts S_{11}, S_{12}, S_{21}, S_{22}, and call $\mathsf{DeleteCross}$ for every pair of parts (there's four pairs).

  So we set $S_1 \leftarrow S_{11} \cup S_{12}$ and $S_2 \leftarrow S_{21} \cup S_{22}$, where $|S_{1r}| = |S_{2p}| = |S_1|/2$ for $r, p \in \{1, 2\}$. Then for each choice $r, p \in \{1, 2\}$, we set

  $$U \leftarrow \mathsf{DeleteCross}(S_{1r}, S_{2p}).$$

  (So we're doing a recursive call with the same procedure.) After we do that, we may have messed up the inverse, so we have to fix the inverse — we run $\mathsf{Update}(N, U)$ on $S \times S$. So we fix all the entries of the inverse, but only in the $S \times S$ portion (because this is the only part we care about.)

  After we've done that, now the inverse is correct again, so we do it again.

  We also have to keep track of something — we start with $R = \emptyset$, and every time we do this, we set $R \leftarrow R \cup U$. And in the end, we return R.

So R is the total set of removed edges in the full recursive call; and we take the union of them and we're done. After each recursive call we have to update the inverse because we may have messed it up, but then it will remain correct throughout the loop?

What is the runtime of this thing? We're going to call it $T_{\mathsf{cross}}(n)$; what's the recurrence? We get

$$T_{\mathsf{cross}}(n) \leq 4T_{\mathsf{cross}}(n/2) + n^\omega$$

(where $n = |S|$). The n^ω is the update step, which takes time $|S|^\omega$, and then I did four recursive calls. We have $\omega \geq 2$, so this solves to $T_{\mathsf{cross}}() \leq \widetilde{O}(n^\omega)$. (If $\omega = 2$ then there's a $\log n$ factor, which is why we use $\widetilde{O}$.)

So this entire crossing procedure is pretty cheap. But we're not done — what we need to do is the higher procedure where we have a single set S, and we're splitting it into two halves. So let's do that.

Now we'll define $\mathsf{DeleteWithin}(S)$. Here we assume $N(S,S)$ is correct.

For the base case, what happens if $|S| = 1$? Then there's no edges in $S \times S$, so we just return $\emptyset$. (This procedure is supposed to return some set of edges $U \subseteq S \times S$ that can be removed such that the remaining graph still has a perfect matching; $S \times S$ has no edges, so we don't return anything.)

Otherwise $|S| \geq 2$; we'll assume for convenience that $|S|$ is a power of 2.

Then we're going to split S into two parts $S \leftarrow S_1 \cup S_2$, just as in our picture.

The first thing we do is set $U_1 \leftarrow \mathsf{DeleteWithin}(S_1)$ — we find all the edges with endpoints in S_1 that can be removed. And then we have to fix what we broke, so we update N by running $\mathsf{UpdateN}(N, U_1)$ on $S \times S$.

Then we do the same thing in the other half — we set

$$U_2 \leftarrow \mathsf{DeleteWithin}(S_2)$$

and run $\mathsf{UpdateN}(N, U_2)$ on $S \times S$.

Finally, we set $U_3 \leftarrow \mathsf{DeleteCross}(S_1, S_2)$ and run $\mathsf{UpdateN}(N, U_3)$.

Finally, we return $U_1 \cup U_2 \cup U_3$.

So here we find what edges in $S_1 \times S_1$ we can delete; then we find all the edges in $S_2 \times S_2$ that we can delete (after we've deleted these ones); and then we find all edges in $S_1 \times S_2$ we can delete (after we've deleted these guys) — we're updating the inverse, so we can check iteratively which edges can be removed. And in the end, we just return everything that can be removed.

Every time we call DeleteWithin or DeleteCross, we're fixing the appropriate submatrix — exactly the one we'll need in the rest of the computation recursively. This is a bit tricky, but that's roughly what's happening.

Now, what is the runtime of this? We get

$$T_{\mathsf{in}}(n) \leq 2T_{\mathsf{in}}\left(\frac{n}{2}\right) + T_{\mathsf{cross}}\left(\frac{n}{2}\right) + n^\omega,$$

where this n^ω is the update time. And we know by our previous analysis that $T_{\mathsf{cross}}(n) = \widetilde{O}(n^\omega)$. So we get $T(n) = 2T(n/2) + \widetilde{O}(n^\omega)$, and therefore

$$T_{\mathsf{in}}(n) \leq \widetilde{O}(n^\omega).$$

And now for the final algorithm:

Algorithm 16.6 (Harvey matching)

Set T to be a random evaluation of the Tutte matrix.

Compute $N \leftarrow T^{-1}$.

Set $U \leftarrow \mathsf{DeleteWithin}(V)$.

Return $E \setminus U$.

So you take your Tutte matrix and compute its inverse (now that's completely correct, so the promise at the beginning of DeleteWithin is true — $N[V,V]$ is correct). Then in the recursive calls, everything will be correct (we keep updating everything we need). This will return all the edges that can be removed such that the remaining thing thing still has a perfect matching; and that means $E \setminus U$ must be a perfect matching! (If it had any extra edges, those could be removed and we'd still have a perfect matching.)

So this gives an n^ω time algorithm.

To Virginia, this is very surprising — she thought finding an edge to add to a perfect matching would be easier. But it turns out it's easier to get rid of the ones you don't need, which is very interesting.

Student Question. *Could there be efficient algorithms to find a perfect matching that only use the perfect matching detection algorithm as a black box?*

Answer. With the same runtime? The problem is that you need to analyze the black box in terms of the size of the subgraph you're calling it on. You're asking if you can do this without white-box updating the inverse — if you can get just a reduction to detection? It would have to be a very interesting reduction; Virginia doesn't know how to do it. But it's a good question. She doesn't know if you can get a nice fine-grained reduction where you care about the runtimes. Because here we really rely on these formulas for updating the inverse.

Remark 16.7. There's a few things about this algorithm. It's the best-known algorithm for dense general graphs, and it's randomized; the best-known deterministic algorithms are much slower. There's $m\sqrt{n}$, but when $m = n^2$ that's $n^{2.5}$, which is slower. And they're also much more complicated. This thing here is a bit delicated, but it's a very simple, short algorithm; while the combinatorial algorithms that are deterministic are much more complicated and use blossoms and are really messy. There's been lots of work in trying to find a deterministic algorithm that's not so messy, and people have failed. We don't know how to derandomize this — we're using Schwarz–Zippel, and derandomizing polynomial identity testing seems difficult. So this is still a big open problem. And we don't know how to use flow techniques for general graphs, because of odd cycles.

The paper list will be released today; Virginia will publish a list, and every paper that will be taken will be crossed out (but there will be a long list, so hopefully there won't be too many collisions).

§17 April 10, 2025 — Baur–Strassen theorem and applications

PS4 is due today sometime tonight, and PS5 will be out tonight. Virginia has put up the paper list; we can go through it and let her know what paper we want. It's first-come first-serve, so we're encouraged to email earlier rather than later; we have until the 17th, but if you really want some paper, you should do it earlier.

Today we'll talk about the Baur–Strassen theorem, which is a very cool theorem about programs that compute rational functions. Then we're going to use it to give an algorithm for the shortest cycle problem.

§17.1 Straight-line programs

In order to do this, we'll first talk about the types of algorithms we'll be talking about today. We'll be looking at algorithms that compute rational functions. Suppose we have a field $\mathbb{F}$ (which could be $\mathbb{Q}$, $\mathbb{R}$, $\mathbb{F}_p$, or so on). And we want to compute some rational function over it — so we have a rational function $F(x_1, \ldots, x_n)$ over $\mathbb{F}$. In general, your algorithm could be very complicated. But we're going to look at a particular type of algorithm, called an *arithmetic circuit* or *straight-line program* (SLP).

Whatever field we pick is going to have some (binary) operations — addition, multiplication, division, and so on. We're not going to include subtraction — you can simulate it by multiplying by -1 and then adding. So we have these binary operations. And what is a SLP?

Definition 17.1. A *straight-line program* is a sequence of instructions $g_1, g_2, \ldots, g_s$ (where s is the *size* of the SLP) of the following form:

(1) $g_i \leftarrow g_j \odot g_k$ for $j, k < i$ (where $\odot$ is some field operation).

(2) $g_i \leftarrow x_a \odot x_b$ (where x_a and x_b are variables).

(3) $g_i \leftarrow g_j \odot x_a$ where $j < i$.

(4) $g_i \leftarrow g_j \odot c$, where $c \in \mathbb{F}$ is a scalar and $j < i$.

(5) $g_i \leftarrow x_a \cdot c$.

So we have a sequence of operations where g_i is some binary operation between things that have appeared before it (the letter g is for 'gate'), or variables (which you think of as appearing all the way on the left as input). And of course the scalars are always around.

What does it mean for a SLP to *compute* F? If F has one output, then we assume g_s is supposed to be that output — i.e.,

$$g_s = F(x_1, \ldots, x_n).$$

Otherwise, if there's multiple outputs, then you'll have specified some set of g_i to be the output gates.

Let's give an example:

Example 17.2

Suppose I want to compute the function $F(x_1, x_2, x_3) = x_1x_2 + x_2^2x_1/x_3$.

So this is a single-output function. And here's a SLP for it:

- $g_1 \leftarrow x_1 \cdot x_2$.
- $g_2 \leftarrow g_1 \cdot x_2$.
- $g_3 \leftarrow g_2/x_3$.
- $g_4 \leftarrow g_1 + g_3$. (And that's your output.)

(Then g_2 stores $x_1x_2^2$, g_3 stores $x_1x_2^2/x_3$, and g_4 stores the right output.) The size of this program is 4 — there's only 4 gates.

Why do we call it a circuit as well? Well, you can represent it as a circuit. Here, x_1, x_2, and x_3 are your inputs at the bottom. And then you have gates. Gate g_1 is a multiplication gate (labelled with $\cdot$), and its inputs are x_1 and x_2. Gate g_2 is also a multiplication gate, and its inputs are g_1 and x_2 (drawn with arrows from both into g_1 and g_2). And g_3 is a division gate, whose inputs are g_2 and g_3 (with output). And g_4 (the output) is an addition gate, whose inputs are g_1 and g_3 (we draw gate type by circling the operation). So there's your circuit.

There's a one-to-one correspondence between SLPs and circuits (circuits are DAGs, and what we've described is the topological order of the DAG), with fan-in 2.

§17.2 The Baur–Strassen theorem

These are very special types of algorithms, and we're interested in how efficient they are.

Definition 17.3. For a rational function F, let $T(F)$ be the minimum size of a SLP computing F.

(We think of the field as fixed.)

So for the function above, probably the minimum size is 4 (Virginia doesn't see how to do better).

Question 17.4. For various particular functions F, what is $T(F)$?

The Baur–Strassen theorem gives a very interesting statement that if you have a very good bound on your minimum-size program for a function, then you can get a very similar-sized program for also computing all the partial derivatives of the function.

Theorem 17.5 (Baur–Strassen)

Let F be a rational function (with a single output), and let F' be the multi-output function whose outputs are $\frac{\partial F}{\partial x_1}, \dots, \frac{\partial F}{\partial x_n}$. Then $T(F') \leq 6 \cdot T(F)$.

This says if you have an arithmetic circuit of size $T(F)$ for computing F, you can construct an arithmetic circuit of at most 6 times that size which also computes all the partials. This is very useful.

Remark 17.6. Let's get back to what computing means. Computing doesn't mean it'll output your full polynomial with all its coefficients; it means it'll give you a representation such that for any values x_i you plug in, you can run this circuit and get the right output.

This should look surprising. Not only is it surprising, but the proof is surprisingly simple, so we're actually going to prove it.

Proof. We have our function F; and let $g_1, \dots, g_s$ be the optimal SLP of it (where $s = T(F)$). This means g_s is the output, so $g_s = F(x_1, \dots, x_n)$.

We're going to define a series of functions according to this topological order; and then we'll use induction on these functions to get all the partials. So we write out $g_1, g_2, g_3, \dots, g_{i-1}, g_i, \dots, g_s$ spaced out in a line.

Let's look at the part $g_i, \dots, g_s$ of the sequence of operations. You can think of this part as a SLP itself, whose input is $g_1, g_2, \dots, g_{i-1}$, and also $x_1, \dots, x_n$. So this portion $(g_i, \dots, g_s)$ computes some function $F^{(i)}$ on inputs $x_1, \dots, x_n, g_1, \dots, g_{i-1}$. All of these operations, by the way we defined them, are operations on scalars, the variables, and all the gates that are before — so this in particular is a SLP for this function $F^{(i)}$ with output g_s, but on these variables.

We'll also define

$$F^{(s+1)}(x_1, \dots, x_n, g_1, \dots, g_s) = g_s.$$

(This is kind of just the last function, which takes all these inputs and spits out the output.)

We'll assume we've already computed the partials for this part $F^{(i)}$ — these partials are with respect to x_1, $\dots$, x_n and $g_1, \dots, g_{i-1}$. And then we'll use these partials to also do it for $F^{(i-1)}$.

So the structure of our SLP will be, first we'll compute $g_1, \dots, g_s$, which will give us F. After that, we'll be tacking on operations that compute the partials of $F^{(s+1)}$, $F^{(s)}$, $F^{(s-1)}$, $\dots$, all the way up to the bottom. So the final SLP that we want will be $g_1, \dots, g_s$, and then stuff for computing $\frac{\partial F^{(s+1)}}{\partial \bullet}$, then $\frac{\partial F^{(s)}}{\partial \bullet}$, and so on, and at the very end we'll have $\frac{\partial F^{(1)}}{\partial x_i}$, where $F^{(1)} = F$.

So our goal is, if we're somewhere in the middle and we've computed up to $\frac{\partial F^{(i)}}{\partial \bullet}$, we want to tack on one for $\frac{\partial F^{(i-1)}}{\partial \bullet}$. And we want to say each of these blocks will actually have size at most 5 — each of these will have at most 5 operations. So because there are s of these, we'll be getting $5s$ extra, plus the original s — so we'll get $6s$. So that's the goal.

Let's do the base case. For the base case, the inputs to $F^{(s+1)}$ are all these guys $x_1, \dots, x_n, g_1, \dots, g_s$. But the output only depends on the last one. So in fact

$$\frac{\partial F^{(s+1)}}{\partial g_s} = 1 \quad \text{and} \quad \frac{\partial F^{(s+1)}}{\partial z} = 0 \text{ for } z \neq g_s.$$

Notably, we don't need to do *any* operations for this — we can just output — so this takes us 0 operations.

Now we're going to assume that for $F^{(i)}$, we used at most $5(s+1-i)$ extra operations. So up to this point, I've used at most this many operations (to compute the partials of $F^{(s+1)}$, $F^{(s)}$, $\dots$, $F^{(i)}$). I want to show that I only need to tack on 5 more — to get the partials of $F^{(i-1)}$, we only need at most 5 more operations. So we'll get a total of

$$5(s+1-i)+5 = 5(s+1-(i-1))$$

operations in total. And then if we get that, at the very end when $i=1$, we'll get $5\cdot(s+1-1) = 5s = 5\cdot T(F)$ extra operations, which is what we wanted — so when we get to $i=1$ we'll have added 5 times the size of the original SLP, and together with the original SLP we'll get $6T(F)$.

Now we'll look at how to do this inductive step. In order to do this, we have to look at — what is $F^{(i-1)}$ in terms of $F^{(i)}$? It's exactly the same as $F^{(i)}$, except that it doesn't have g_{i-1} as input — instead of g_{i-1}, you've plugged in the operation that g_{i-1} is performing inside $F^{(i)}$. That's the only difference.

So if $g_{i-1} \leftarrow a \circ b$ (where $a, b \in \mathbb{F} \cup \{x_1, \dots, x_n\} \cup \{g_\ell\}_{\ell<i-1}$ are in the field or are variables or gates), then

$$F^{(i-1)}(\cdots) = F^{(i)}|_{g_{i-1}=a\odot b}$$

(so we're plugging in $g_{i-1} = a \odot b$). This immediately tells us that all the partials of $F^{(i-1)}$ which are not with respect to a or b are the same as $F^{(i)}$ — for every $z \neq a, b$, we have

$$\frac{\partial F^{(i-1)}}{\partial z} = \frac{\partial F^{(i)}}{\partial z}.$$

The only difference is that now we need the derivatives with respect to a or b. But for that, we can use the chain rule, which says

$$\frac{\partial F^{(i-1)}}{\partial a} = \frac{\partial F^{(i)}}{\partial a} + \frac{\partial F^{(i)}}{\partial g_{i-1}} \cdot \frac{\partial g_{i-1}}{\partial a},$$

and the same for b. So we only need to compute two new partials, and th rest we inherit from $F^{(i)}$ and we don't need to do extra computation because of the way we tacked on things (up to $F^{(i)}$, in our circuit, we have all the partials of $F^{(i)}$; so we've already inherited all the partials that are the same, and we only need to fix (at most) two of them).

We can also do the same for b. So there's these two partials that we need to fix. And we'll look at all the operations we had, and see how many operations we need to perform. For simplicity we're not going to do all of them; we'll just look at two (and you can see the notes for the rest). We're going to do a simple one and then a complicated one.

Case 1 ($g_{i-1} = z_1 + z_2$, where $z_1, z_2 \in \{x_j\} \cup \{g_\ell\}_{\ell<i-1}$). So if it's just an addition gate, then we have to do the following partials — we have

$$\frac{\partial F^{(i-1)}}{\partial z_1} \leftarrow \frac{\partial F^{(i)}}{\partial z_1} + \frac{\partial F^{(i)}}{\partial g_{i-1}} \cdot 1,$$

and it's the same for z_2. How many operations do we need to compute both of these? It's 2 (in total) — the addition in the first, and the addition in the second. (Here we assumed $z_1 \neq z_2$.)

A more interesting one:

Case 2 ($g_{i-1} = z_1/z_2$ where $z_1 \neq z_2$). Here they're actually different — we have

$$\frac{\partial F^{(i-1)}}{\partial z_1} = \frac{\partial F^{(i)}}{\partial z_1} + \frac{\partial F^{(i)}}{\partial g_{i-1}}/z_2,$$
$$\frac{\partial F^{(i-1)}}{\partial z_2} = \frac{\partial F^{(i)}}{\partial z_2} + \frac{\partial F^{(i)}}{\partial g_{i-1}} \cdot \left(-\frac{z_1}{z_2^2}\right).$$

Now we claim that we can compute both of these with 5 operations. Why? The point si that $(\partial F^{(i)}/\partial g_{i-1})/z_2$ is reused. So we do:

- $a \leftarrow (\partial F^{(i)}/\partial g_{i-1})/z_2$.
- $b \leftarrow a/z_2$.
- $c \leftarrow b \cdot z_1$.
- $d \leftarrow c \cdot (-1)$.
- $e \leftarrow (\partial F^{(i)}/\partial z_1) + a$.
- $f \leftarrow (\partial F^{(i)}/\partial z_2) + d$.

Why is this 6? It was supposed to be 5...

Actually the point is that we have $g_{i-1} = z_1/z_2$, which we haven't used. So instead of dividing by z_2 and multiplying by z_1, we can multiply by g_{i-1}, and that's how we save.

- $a \leftarrow (\partial F^{(i)}/\partial g_{i-1})/z_2$.
- $b \leftarrow a \cdot g_1$.
- $d \leftarrow b \cdot (-1)$.
- $e \leftarrow (\partial F^{(i)}/\partial z_1) + a$.
- $f \leftarrow (\partial F^{(i)}/\partial z_2) + d$.

And actually this turns out to be the worst case — every other case is simpler (multiplication gets 4; multiplying by a scalar is really cheap; and so on).

So this is the full proof, basically. Once you see how to get all the derivatives, you just tack them on one by one; and in the end you get $6s$ is your total length. □

So basically this means if you have a SLP for evaluating any (single-output) function you want, you can get one of roughly the same size that also gets you all the partials of the function. This is extremely useful. We'll see one application to graph algorithms, but it's useful all over the place.

> **Remark 17.7.** On the problem set, we'll give you a little exercise — that if you could do the same thing and get all the second partials with constant overhead, then you'd have shown $\omega = 2$.

§17.3 The shortest cycle problem

Next, we'll see an application of this to the shortest cycle problem in graphs.

> **Algorithm 17.8** (Shortest cycle)
> - **Input:** A directed graph $G = (V, E)$ with weights $w : E \to \{1, 2, \ldots, M\}$.
> - **Output:** Find the cycle in G of minimum weight.

So we have a directed graph with weights on the edges (which are positive integers bounded by some M). The weight of a cycle is the sum of weights of its edges.

We know you can solve this using APSP — you compute for every vertex the shortest path to every other vertex $i \rightsquigarrow j$; and then if there's also an edge $j \to i$, this will give you a shortest cycle. And then you minimize over all pairs (i, j).

So using APSP, you can solve this problem 'fast.' What does 'fast' mean? You can get $n^3/\exp(\sqrt{\log n})$ using APSP. And we also know that Shortest Cycle is equivalent to APSP in a certain sense — APSP is in $O(n^{3-\varepsilon}\mathsf{polylog}(M))$ time if and only if Shortest Cycle is in $O(n^{3-\varepsilon'}\mathsf{polylog}(M))$ time, and vice versa. So we won't be able to get algorithms of this form.

What we're going to get is:

Claim 17.9 — We can solve SC in $\widetilde{O}(M \cdot n^\omega)$ time.

So instead of having a polylog dependence in weights, we'll get a pseudopolynomial overhead. When the weights are very small (e.g., constant), you can solve this problem faster; but as they get bigger and bigger this overhead kills us, so we might as well use the $n^3/\exp(\sqrt{\log n})$ algorithm.

There are many proofs of this claim; you don't need Baur–Strassen, but it gives a cleaner proof.

The way we'll do this is define a matrix with variables (a symbolic matrix), similarly to with perfect matching algorithms. We'll look at its determinant, which is some polynomial, and we'll discuss what the determinant means in this case. In order to do this, we'll define something about polynomials.

Definition 17.10. Suppose P is a multivariate polynomial, and let y be a special variable. Then we define $d^*(P)$ to be the smallest d such that the coefficient of y^d is a nonzero polynomial.

Example 17.11

Suppose $P(x_1, x_2, y) = x_1^2 + x_2 y^{10}$. Then $d^* = 0$, because the coefficient in front of y^0 is a nonzero polynomial (namely, x_1^2).

Example 17.12

If $P(x_1, y) = x_1 y^7 + x_1^2 y^2$, then $d^*(P) = 2$ (since y^2 is the smallest degree for which we have a coefficient which is a nonzero polynomial).

Student Question. *Could we also get $\widetilde{O}(Mn^\omega)$ by doing a Yuster–Zwick oracle, and then doing n shortest path queries?*

Answer. Yes; that's why we said there's many ways to achieve this runtime. Once you have the Yuster–Zwick oracle, you can do this; the paper using Baur–Strassen gives an alternative way. That paper has many other applications of Baur–Strassen; they're just more complicated.

Definition 17.13. We define $f^*(P)$ as the coefficient in front of y^{d^*} (this is some nonzero polynomial in the rest of the variabes).

Now we're going to start talking about how to get this alternative algorithm for shortest cycle. So G is our given graph. We're going to create a variable for every edge — for every $e \in E$, we'll have a variable x_e. We'll also have an extra variable y (which is special, and different from these).

We'll define a $n \times n$ matrix A where

$$A(i,j) = \begin{cases} x_{(i,j)} \cdot y^{w(i,j)} & \text{if } (i,j) \in E \\ 1 & \text{if } i = j \\ 0 & \text{otherwise.} \end{cases}$$

Example 17.14

Suppose we have the triangle $1 \to 2 \to 3 \to 1$, with edge weights 2, 1, 1. Then the matrix will be

$$\begin{bmatrix} 1 & x_{(1,2)}y^2 & 0 \\ 0 & 1 & x_{(2,3)}y \\ x_{(3,1)}y & 0 & 1 \end{bmatrix}.$$

So we have these polynomials in the entries whenever there's an edge.

What we're going to be interested in is the determinant of this matrix. We're going to claim the following funny thing:

> **Claim 17.15 —** We have that $d^*(\det(A) - 1)$ is exactly the weight of the shortest cycle.

Proof. We had the formula for $\det A$ as

$$\det(A) = \sum_{\sigma \in S_n} (-1)^{\text{sgn}(\sigma)} \cdot \prod_{k=1}^{n} A(k, \sigma(k)).$$

Second, if we look at the portion $(-1)^\bullet \cdot \prod \bullet$ for $\sigma = \text{id}$, we have

$$(-1)^{\text{sgn id}} \cdot \prod_{k=1}^{n} A(k,k) = 1$$

(the sign of the identity is 0, because the sign is the parity of even cycles, and id doesn't have any even cycles; and the $A(k,k)$'s are all 1 by definition). So

$$\det(A) - 1 = \sum_{\sigma \in S_n, \sigma \neq \text{id}} (-1)^{\text{sgn}(\sigma)} \prod_{k} A(k, \sigma(k))$$

(where the sum is over all permutations that are not the identity).

Now, what are the permutations when we look here? They just correspond to *cycle packings* — a node-disjoint set of cycles that cover all the vertices. Why? Like we discussed in the perfect matching lecture, every permutation σ can be viewed as a set of cycles, or a cycle-packing of G — a node-disjoint set of cycles that covers all the vertices. For example, in the above graph there's really two cycle packings. One is (1)(2)(3) (the identity), and the other is (123).

This is because the terms $A(k, \sigma(k))$ are only nonzero on the edges or fixed points of the permutation. So we can rewrite $\det(A)$ as

$$\sum_{\text{cycle packings } \sigma} (\pm 1) \cdot \prod_{C \in \sigma} \prod_{(i,j) \in C} x_{(i,j)} w^{(i,j)}$$

(where $C \in \sigma$ means C is a cycle in your cycle packing). Here the $\pm$ is the sign; and then I'm unwrapping what $A(k, \sigma(k))$ is. So you look at all the edges in your cycle, and A for that edge is just this; and the singleton cycles only have a 1, so we only care about nontrivial cycles in the packing (i.e., non-singletons).

And what is this? We can rewrite it as

$$\sum_{\text{nontrivial cycle packings}} (\pm 1) \cdot \prod_{C \in \sigma} y^{w(C)} \prod_{e \in C} x_e.$$

This is a sum of terms whose degree in y is a sum of cycle weights, in a cycle packing. Because of this, the minimum degree must be the weight of a single cycle — if I pick a cycle packing that has more than one nontrivial cycle, then I can replace the second cycle with the corresponding singletons, and I get a better weight. So the minimum degree of this polynomial in y is exactly the weight of the minimum cycle, and that's how we get the claim. □

(We subtracted 1 because otherwise we'd have the trivial cycle packing with all singletones; then d^* would be 0, and this wouldn't be useful to us.)

Now let's look at what $f^*(\det A - 1)$ is. You'll have a sum over all the shortest cycles, of a ± 1 times the product of the edges on the cycle (or rather, their corresponding x's); so we get

$$f^*(\det A - 1) = \sum_{w(C)=d^*} \pm 1 \cdot \prod_{e \in C} x_e$$

(where the sum is over all cycles in C^* whose weight is d^*, i.e., shortest cycles; the ± 1 depends on the parity of their length).

What we'll claim (very similarly to the matching approach) is that if I pick random values for the x_e's, then with high probability this will remain nonzero. So by Schwarz–Zippel, if we pick random values for all x_e from some $\mathbb{F}_p$ (where p is large enough, e.g. $p > n^2$) and plug them in, then

$$f^*(\det A - 1) \neq 0$$

with high probability (once we plug in the values of the x_e variables, we had a nonzero polynomial, so it'll remain nonzero with high probability).

This means if I plug in random variables for the x_e variables (into A) but leave y the same (i.e., as a variable), then d^* will remain the same with high probability.

Now once I've plugged in these random values, I have a matrix over $\mathbb{F}_p$ whose entries are just polynomials in a single variable y. So what remains is a matrix A with entries that are some monomial in y (basically, y to the weight of some edge, or 0). So A (evaluated) has entries $v_e \cdot y^{w(e)}$ in the edge entries (instead of x_e you have values from the field).

Now all we need to do is figure out how to evaluate the determinant of such a matrix — which is symbolic, but only in one variable.

And it turns out there is a way to do this: If you think about multiplying matrices whose entries are polynomials in a single variable, then we can multiply them using FFT, and the overhead is just the degrees. A similar thing happens with the determinant: If you have a symbolic matrix over a single variable whose degree is at most d, then you can evaluate the degree in $d \cdot n^\omega$ time.

Theorem 17.16 (Storjohann)

Let A be a matrix such that $A(i,j)$ is a polynomial over y over some field $\mathbb{F}$ of degree at most M. Then $\det(A)$ can be computed by a SLP of size $\widetilde{O}(Mn^\omega)$.

So not only is there an *algorithm*, there's actually an arithmetic circuit.

Now we're going to combine this with Baur–Strassen; and we'll be able to get not only the weight of the shortest cycle, but actually the shortest cycle as well.

For the weight of the shortest cycle, how do we do it (given this theorem)? What this tells me is that — what it means to compute $\det A$ is that for any values of the x-variables, I can give you the coefficient in front of each power of y. So this SLP gives the coefficients of y^c for all $c \leq Mn$ (in our case). We can just look at all of them, and find the smallest c that has a nonzero coefficient.

So to get the weight, we compute $\det(A) - 1$ and find d^*, just by looking at the coefficients; wth high probability we know d^* remains unchanged when we plug in random values. This will take us $\widetilde{O}(Mn^\omega)$ time (from computing the determinant and scanning the coefficient).

Now, the second thing is that we can find an edge $e \in C$ in the same time. How? Combining these two theorems, we know that we can get not just the determinant, but also all the partials — via Baur–Strassen, there is a SLOP of size $\widetilde{O}(Mn^\omega)$ such that for every e, it computes

$$\frac{\partial(\det(A) - 1)}{\partial x_e}.$$

The reason why these x_e's are variables is that the input to your matrix is basically the $v_e w^{y(e)}$ (where the v_e's are x_e's evaluated).

Now that we have this, we can look at f^*. We know what d^* is, so we can get f^* evaluated frmo Storjohann and BS. Now if we look at its partial with respect to som evariable, it'lll be 0 if that variable doesn't appear in the shortest cycle, and nonzero if it does appear. So from here, we get that

$$\frac{\partial f^*(\det A - 1)}{\partial x_e} = \begin{cases} 1 & e \text{ is in some shortest cycle} \\ 0 \text{ whp} & \text{otherwise.} \end{cases}$$

Student Question. *What if you have multiple disjoint shortest cycles?*

Answer. That won't be an issue — all we'll need is one edge.

This will give me some edge that's on the shortest cycle (maybe multiple). Now what do I do? I just run Djikstra's algorithm. Suppose it tells me $e = (i, j)$ is on some shortest cycle. Then I run Djikstra's algorithm from j. It'll give me the shortest path from j to i, and then we tack on this edge and that's a shortest cycle. (So I only need a single edge on the shortest cycle, which I get from the partials.) And that's only n^2 extra overhead; and this gives me the full cycle.

§18 April 15, 2025 — Matrix multiplication and bilinear problems

Now begins the part of the course where we'll actually design matrix multiplication algorithms. We're going to define a bunch of formalism today that we'll use, but we'll see some intuition for why we're doing this.

§18.1 Arithmetic circuits

The first thing we'll talk about is something we talked about last time — arithmetic circuits, or straight-line programs.

So we had some underlying field $\mathbb{F}$ (e.g., $\mathbb{R}$), and we were interested in computing rational functions over various variables over this field. The types of algorithms we looked at were these algorithms that had a bunch of operations, one after another. And the operations we allowed were binary operations of the form $a \odot b$ — you had gates $g_1, \ldots, g_s$ and inputs $x_1, \ldots, x_n$, where each gate was of the form

$$g_i \leftarrow g_j \odot g_k$$

where $\odot \in \{+, -, *, /\}$ (this time we'll allow subtraction as well), and $j, k < i$ (so we use previously computed values). Last time we allowed the inputs to the binary operation to also be scalars; today we're going to also allow things like $\lambda \cdot g_j$ and $\lambda + g_j$ for $\lambda \in \mathbb{F}$ (or g_j/λ, but that's the same as multiplication, so we'll ignore it).

So these are the types of operations we'll allow. Today we'll talk about what it means to measure the cost of an arithmetic circuit. Last time we measured cost by the number of gates; so that's one way to measure cost.

But another cost, which we'll talk about today, is the number of $*$ and $/$ operations with gates — things like $g_j * g_k$ and g_j/g_k. This basically says that multiplication and addition with scalars are free; and so are addition and subtraction of previous values. So these two operations have cost 1, and all others cost 0. This is called the *Ostrowski cost model*.

More formally, let's say $\mathcal{F}$ is some set of rational functions $\{f_1, \ldots, f_k\}$ over variables $x_1, \ldots, x_n$ over $\mathbb{F}$.

Definition 18.1. We define the *Ostrowski cost* $C^{*/}(\mathcal{F})$ as the minimum Ostrowski cost of an SLP computing $\mathcal{F}$.

So you're allowed to put as many scalar operations as you want, as many additions and subtractions of previous values; those are completely for free. The only thing you care about is the number of products and divisions.

We'll see why this is interesting; but this is what we're going to care about.

§18.2 Strassen's algorithm

Why is this interesting?

Strassen at some point was thinking about matrix multiplication. People believed it's a very difficult problem and requires circuits of size n^3. So he looked at 2×2 matrix multiplication, which is the problem

$$\begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{bmatrix}, \begin{bmatrix} y_{11} & y_{12} \\ y_{21} & y_{22} \end{bmatrix} \to \begin{bmatrix} z_{11} & z_{12} \\ z_{21} & z_{22} \end{bmatrix}.$$

We get additions and subtractions for free; and in the definition, you have 8 multiplications. So you can get Ostrowski cost 8. He tried for a long time to show that you can't get fewer than 8. But he couldn't prove it, because it turned out not to be true! You can get less than 8. Specifically, you can compute the following 7 products:

- $P_1 \leftarrow (x_{11} + x_{22}) \cdot (y_{11} + y_{22})$.
- $P_2 \leftarrow (x_2 + x_{22}) \cdot y_{11}$.
- $P_3 \leftarrow x_{11} \cdot (y_{12} - y_{22})$.
- $P_4 \leftarrow x_{22} \cdot (y_{21} - y_{11})$.
- $P_5 \leftarrow (x_{11} + x_{12}) \cdot y_{22}$.
- $P_6 \leftarrow (x_{21} - x_{11})(y_{11} + y_{12})$.
- $P_7 \leftarrow (x_{12} - x_{22})(y_{21} + y_{22})$.

And it turns out that you can express every output as a linear combination of these.

Example 18.2

We have $z_{21} \leftarrow P_2 + P_4$ — we have

$$x_{21}y_{11} + x_{22}y_{11} + x_{22}y_{21} - x_{22}y_{11},$$

and the appropriate terms cancel and you get what you need.

You do many more additions, but it turns out that doesn't matter.

Why doesn't it matter? Let's look at this 2×2 matrix multiplication algorith, and see what happens with $n \times n$ matrices. So we have a $n \times n$ matrix X and another Y; let's say n is a power of 2. You can split them into submatrices of size $n/2 \times n/2$, and when you look at the product, it's also an $n \times n$ matrix Z, which you can *also* split up. And now you can write

$$X = \begin{bmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{bmatrix},$$

and similarly with Y and Z. And the relationship between these subblocks is exactly the same as with the x, y, and z's, except that these are no longer entries but now matrices.

And then you can view the 2×2 multiplication algorithm, except that instead of numbers you have matrices. Then with the additions you're summing matrices — that's extremely cheap (it takes time $O(n^2)$, since you can do it pointwise, which is linear in the size of the input), so it's essentially as if they're free.

So your algorithm is, you take both your input matrices, split them into pieces, follow this algorithm — compute your linear combinations in linear time, and then do the products *recursively* (when you do the recursion, you do exactly the same splitting-in-half things until you have single entries). Then you'll get a recurrence of the form

$$T(n) = 7T(n/2) + O(n^2)$$

(where $O(n^2)$ is to form the linear combinations like $X_{11}+X_{22}$; after you've formed them you do the products recursively; and then it's also $O(n^2)$ time to get the Z_{ij}'s by taking linear combinations of the P_i's). So you have this recurrence, adn it solves to

$$T(n) = O(n^{\log_2 7}) \approx O(n^{2.81}).$$

This was the first truly subcubic algorithm for matrix multiplication. And it really relies on a small 2×2 algorithm of a specific form — you take linear combinations of one matrix, then another, and then do products. The reason you have to do it this way — you can't mix them up — is because you need the product to make sense for matrices, not just numbers (and matrix multiplication is not commutative, so you have to respect order).

This is the reason we don't care about addition and multiplication by scalars — for matrices, they're linear time, so they're essentially free. This is why the Ostrowski model makes sense.

The goal for the next several lectures is to expand on this recursive paradigm of getting matrix multiplication algorithms — in the beginning, we're just looking at the base case being this 2×2 matrix multiplication algorithm, but later we're going to use some more complicated things to do recursion. So we're going to build up towards that.

§18.3 Quadratic and bilinear problems

Now we're going to define the type of problems we want to solve. Matrix multiplication is one of a type of problem — it's a *quadratic* problem, and also a *bilinear* problem. So we're going to define those two things.

We'll have our field $\mathbb{F}$ (an underlying field of scalars), and also variables $x_1, \ldots, x_n$. Say $\mathcal{F} = \{f_1, \ldots, f_K\}$ is a set of rational functions we want to compute.

Definition 18.3. We say $\mathcal{F}$ is a set of *quadratic forms* if for every $k \in [K]$, f_K is of the form

$$f_k = \sum_{i,j=1}^{n} t_{ijk} \cdot x_i \cdot x_j.$$

So there's some coefficients such that you can form each one of the functions you're computing as a linear combination of quadratic things. (This is an order-3 tensor; we'll discuss these in a little bit.)

Notably, matrix multiplication is one of these, and we'll express it in this form in a little bit.

Definition 18.4. We say $\mathcal{F}$ is a set of *bilinear forms* if the variables can be partitioned into $x_1, \ldots, x_n$ and $y_1, \ldots, y_m$, and there exist coefficients such that

$$f_k = \sum_{i=1}^{n}\sum_{j=1}^{n} t_{ijk} x_i y_j.$$

Definition 18.5. A *bilinear problem* is, given $x_1, \ldots, x_n$, $y_1, \ldots, y_m$, compute $\{f_1, \ldots, f_K\}$.

So you have x-variables and y-variables, and you have some coefficients t_{ijk}, which define your problem. Given these things that define the problem, your bilinear problem is to just compute these values.

§18.4 Examples

We'll see two examples in this formulation — matrix multiplication and polynomial multiplication. Then we'll see two other ways to represent the problem, which might be nicer depending on the problem; and then we'll discuss SLPs to compute these things.

Example 18.6

In matrix multiplication, we're given matrices, which we can view as vectors of length n^2; say we're given X (which is an $N \times M$ matrix) and Y (which is a $M \times K$ matrix). We can view X as an $N \cdot M$ length vector, and Y as a length $M \cdot K$ vector. And we need to say, what are the coefficients $t_\bullet$ that define the outputs? The output is $z_{ij} = \sum_\ell x_{i\ell} y_{\ell j}$ (the vectors are indexed by pairs of indices).

So we define the *matrix multiplication tensor* as

$$t_{(i,\ell),(\ell',j),(i',j')} = \begin{cases} 1 & \text{if } \ell = \ell',\ j = j',\text{ and } i = i' \\ 0 & \text{otherwise.} \end{cases}$$

(I have to consider all possible triples of indices of my input).

Then when you write out the definition and plug in 1's and 0's for the coefficients, you'll get exactly this — we have

$$\sum_{(i,\ell),(\ell',j),(i',j')} t_{(i,\ell),(\ell',j),(i',j')} x_{i\ell} y_{\ell' j},$$

the $t_\bullet$ will only be 1 when $i = i'$, $j = j'$, and $\ell = \ell'$, so this will be exactly $\sum_\ell x_{i\ell} y_{\ell j}$.

This tensor is defined in a big space — $\mathbb{F}^{(NM)\times(MK)\times(NK)}$ — but it's only 1 in some parts, the ones where the indices match up.

For various reasons we'll see in a little bit, actually the tensor of matrix multiplication is defined so that the output is the *transpose* of the matrix product. So actually:

Definition 18.7. The $N \times M \times K$ matrix multiplication tensor $\langle N, M, K\rangle \in \mathbb{F}^{(NM)\times(MK)\times(KN)}$ is defined by

$$\langle N, M, K\rangle_{(i,\ell),(\ell',j),(j',i')} = \begin{cases} 1 & \text{if } i = i',\ j = j',\ \ell = \ell' \\ 0 & \text{otherwise.} \end{cases}$$

So NM corresponds to the first matrix, MK the second, and KN the *transpose* of the output. The only thing is I flipped the entries of the third thing; this is so it rotates nicely (and we'll see why in a bit, when we define a different representation of these things).

So this is matrix multiplication as a bilinear problem.

Now we're going to define polynomial multiplication over a single variable.

Example 18.8

Suppose we're given two polynomials $\sum_{i=0}^{n} a_i r^i$ and $\sum_{j=0}^{m} b_j r^j$ (so we have a degree-n polynomial and degree-m polynomial of a single variable), and we want to compute a third polynomial $\sum_{k=0}^{m+n} c_k r^k$ which is their product. The relationship is

$$c_k = \sum_{i \leq k} a_i \cdot b_{k-i}.$$

So this is our bilinear problem. And we can write it in the above form, where

$$t_{ijk} = \begin{cases} 1 & \text{if } k = i + j \\ 0 & \text{otherwise.} \end{cases}$$

These are two of the most famous examples, but there are many other problems you might want to compute that have this form.

§18.5 Trilinear representations

Now, to see a diferent way of thinking about it, if you open up a paper about these topics (matrix or polynomial multiplication or so on), you might see something called a *trilinear representation* of these problems. So we'll define that and then rewrite these examples in that form. And then we'll do a pictorial representation (which Virginia likes, because it's nice and visual).

You have your variables $x_1, \ldots, x_n$ and $y_1, \ldots, y_m$, and then you have these coefficients t_{ijk} (for $k = 1, \ldots, K$).

Definition 18.9. The *trilinear representation* is $\sum_{ijk} t_{ijk} x_i y_j z_k$.

So we have a trilinear polynomial. The x and y are your inputs, and z kind of represents the output. But how does it represent the output? The kth output is just the coefficient in front of z_k. Note that z_k is *not* the output — the output is the *coefficient* in front of z_k, which is some bilinear polynomial. So your total output is a set of K bilinear polynomials.

Now we're going to write the trilinear representations of matrix and polynomial multiplication, and we'll see why we flipped the output to be the transpose.

Example 18.10 (Matrix multiplication)

The trilinear representation of matrix multiplication is

$$\sum_{i,\ell,j} x_{i\ell} y_{\ell j} z_{ji}.$$

So this is matrix multiplication in trilinear form — for any particular (i, j), I look at the coefficient in front of z_{ji}, and that's $\sum_\ell x_{i\ell} y_{\ell j}$. I flipped j and i because now this is completely symmetric. And there was no particular reason to pick z as the output — I could have picked x or y and had the same sort of theory. Next time we'll see that by flipping which one is the output, matrix multiplications of dimensions N, M, and K — if you get an algorithm for this, then you can get an algorithm for any permutation of the dimensions. This is weird and wild, but it's because matrix multiplication is extremely symmetric in this way.

Example 18.11 (Polynomial multiplication)
The trilinear representation of polynomial multiplication is $\sum_{i,j} a_i b_j c_{i+j}$.

This is because the coefficient in front of c_k is just $\sum_{i \le k} a_i b_{k-i}$, which is exactly polynomial multiplication.

Remark 18.12. If you've seen the 3-sum problem, this kind of relates to it.

§18.6 Matrix/pictorial representation

Now we'll see a different representation, which is useful in some other ways. This is kind of a matrix representation. It's particularly useful in the case where the coefficients t_{ijk} are either 0 or 1, as in problems like these.

YOu write down a matrix. The columns are labelled by the x-variables — you have columns labelled x_1, $\dots$, x_n. And you label the rows by the y-variables y_1, $\dots$, y_m. (We go up and left-to-right, though you can go a different way.) If I take x_i and y_j, then in the (i,j)th entry, I write all the z_k's with their coefficients that depend on x_i and y_k — so I write

$$\sum_k t_{ijk} z_k,$$

where i and j are fixed.

To see why this is interesting, let's look at what polynomial and matrix multiplication look like — this is useful to identify certain structures.

Example 18.13
Consider polynomial multiplication of degree-2 polynomials.

Here, the inputs are the coefficients of two degree-2 polynomials; so we have 3 coefficients for each.

b_2	c_2	c_3	c_4
b_1	c_1	c_2	c_3
b_0 * c_0	c_1	c_2	
	a_0	a_1	a_2

We can see that all NW-SE diagonals have the same entry. The reason we have really fast algorithms for polynomial multiplication is this structure — you can exploit the fact that diagonals have the same entries, and use roots of unity and fast Fourier transforms, and this is what it exploits.

Example 18.14
Consider 2×2 matrix multiplication.

Here, whenever the second index of an x-entry doesn't match the first index of the y-entry, you'll have a 0 (because the matrix multiplication tensor has to have $\ell = \ell'$ and $j = j'$ and so on). So we immediately get a bunch of 0's.

y_{22}	0	z_{21}	0	z_{22}
y_{21}	0	z_{11}	0	z_{12}
y_{12} z_{21}	0	z_{22}	0	
y_{11} z_{11}	0	z_{12}	0	
	x_{11}	x_{12}	x_{21}	x_{22}

Lots of work, including on group-theoretic algorithms for MM, is to take something that looks like this, and then fudge it so that it looks like the polynomial thing and then use FFT. But as we see, this is not of the same nice diagonal form. Drawing this only really makes sense for small tensors, but it's sometimes useful for seeing what's going on.

§18.7 Restricted circuits for quadratic forms

Now we're back to figuring out the best arithmetic circuit for computing a set of bilinear forms. Strassen looked at quadratic forms more generally. He wondered, do we really need a fancy circuit, or can we figure out there's a very simple type of circuit capturing the best you can do?

He proved that if all you care about are quadratic forms, you don't need divisions — divisions can be completely avoided. And second, you don't need to compute products of things that have degree more than 1. So these restricted types of algorithms give the best complexity we care about.

Theorem 18.15

Let $\mathcal{F} = \{f_1, \ldots, f_K\}$ be a set of quadratic forms (so $f_k = \sum_{i,j=1}^{n} t_{ijk} x_i x_j$ for every $i, j \in [n]$). Suppose that $\mathbb{F}$ is infinite, and that $C^{*/}(\mathcal{F}) = \ell$. Then there exsits a SLP of the following form:

- For $\lambda = 1, \ldots, \ell$, we set
$$P_\lambda \leftarrow \left(\sum_i u_{\lambda i} x_i\right) \cdot \left(\sum_j v_{\lambda j} x_j\right),$$
and for every k, we set
$$f_k = \sum_{\lambda=1}^{\ell} w_{\lambda k} P_\lambda.$$

So we're assuming there exists an arithmetic circuit performing at most ℓ products and divisions of previous gates.

So let's parse this. There exists some AC that performs ℓ products and divisions, then you can transform it into one that only performs ℓ multiplications — and in fact, theyr'e ones of this form, where there are some coefficients u, v, and w (from the field) so that your gates are of this form — take a linear comination of the x variables according to these scalars, and another, and multiply them. The linear combinations have cost 0, because you're multiplying by a scalar and doing summations. So the only thing that matters is the prouct, and you're doing ℓ of them.

And except for the very end, you're not even reusing previous gates.

We won't prove this, but we'll give some intuition on why it works. First of all, you can write
$$\frac{1}{1-x} = \sum_{i=0}^{\infty} x^i.$$
So any sort of division that looks like the LHS, you can think of it as an infinite sum.

But we're computing terms of degree at most 2. So any monomial that has degree more than 2 must eventually cancel — it'll disappear because you don't need it. And if it's going to disappear, why compute it at all? So you can take any division like this and chop it off as a finite sum, and you should be getting the same thing.

Second, if you have products where you're multiplying things with degree more than 2, the same logic applies — any such term has degree more than 2, so you can just chop it off.

Then you have to make this work and so on, but that's the intuition.

He says that for aset of quadratic forms, the arithmetic circuit looks like this.

§18.8 Tensor rank

The next thing we'll do is define rank. To define rank, we'll first mimic what Strassen's algorithm does. And then we'll define the rank of a bilinear form.

For bilinear problems, we can look at an algorithm like this (the above), except the first linear combination is in terms of the x-variables, and the second is in terms of the y-variables.

So for a bilinear problem, we want the minimum number ℓ such that there exists coefficients $u_{\lambda 1}, \ldots, u_{\lambda n}$ and $v_{\lambda_1}, \ldots, v_{\lambda n}$ and $w_{\lambda_1}, \ldots, w_{\lambda_k}$ (where the x-variables are $x_1, \ldots, x_n$, the y-variables are 1 to m, and the ouptuts are 1 to K), given

$$P_\lambda = \left(\sum_i u_{\lambda i} x_i\right)\left(\sum_j v_{\lambda j} y_j\right),$$

we get $f_k = \sum_\lambda w_{\lambda k} P_\ell$.

So wee performed ℓ products, you get your linear combination of the ℓ-products. THese L values will give us λ-values, and then each of the outputs is a linear combination of the products.

This is exactly what's happneing in Strassen — you take a linar conbination of the x-variables and y-variables to get $\ell = 7$ products, and from there you get your linear combinations (the $w_\bullet$'s are your coefficients).

So this is the type of algorithm we look at for bilinear problems; it's basically the same aswhat Strassen showed is optimal for bilinear forms.

Given this, we'll define rank, and then relate it to the optimal thing that Strassen said exists —- we'll basically say that within constant factors, the best algorithm as to look like this.

So what is a rank? We're going to rewrite this in terms of trilinear notation. Remember that was

$$\sum_{ijk} t_{ijk} x_i y_j z_k,$$

where f_k is the coefficient in front of z_k. Now I'm going to write it in terms fo what the decomposition tells me — this is actually

$$\sum_{\lambda=1}^{\ell}\left(\sum_i u_{\lambda i} x_i\right)\left(\sum_j v_{\lambda j} y_j\right)\left(\sum_k w_\lambda z_k\right).$$

Why is that? Well, we look at z_k and its coefficient here is exactly W_k; and in this trilinear expansion we get W_λ.

Now we'll look at what t_{ijk} looks like, in terms of u, v, and w — we have

$$t_{ijk} = \sum_{\lambda=1}^{\ell} u_{\lambda i} v_{\lambda j} w_{\lambda k}.$$

Now, in the world of matrices (which are order-2 tensors), a rank-1 matrix is just the outer product of two vectors. This is the outer produt of *three* vectors.

Definition 18.16. A *rank 1 tensor* is one that can be represented as, given three vectors u, v, and w,

$$t_{ijk} = u_i v_j w_k.$$

We then say this tensor is $u \otimes v \otimes w$.

So a rank-1 tensor is one where you can take three vectors, and express the tensor as their outer product.

Definition 18.17. The rank of a tensor t is the minimum ℓ such that t is a sum of ℓ rank-1 tensors.

Note that $t_{ijk} = \sum_{\lambda=1}^{\ell} u_{\lambda i} v_{\lambda j} w_{\lambda k}$ has rank at most ℓ, since we can write it as

$$t = \sum_{\lambda} u_\lambda \otimes v_\lambda \otimes w_\lambda.$$

Definition 18.18. The rank of a bilinear problem is the rank of its associated tensor.

This is also the smallest arithmetic circuit of the form discussed earlier (i.e., the number of products in it) — any decomposition of your computation is defined by these ℓ triples of vectors $(u_\lambda, v_\lambda, w_\lambda)$ so that you can get this computation; and that's equivalent to expressing the tensor by the sum of ℓ rank-1 tensors given by this triad.

So this is equivalent — if we want an algorithm that looks like this, the only thing we care about is finding the u's, v's, and w's that correspond to the bilinear problem, as a rank-ℓ decomposition. Once we have that, we can transform it into an algorithm; additions and scalar multiplications are free, and we'll be done.

Example 18.19

What Strassen did is he showed $\text{rank}(\langle 2,2,2\rangle) \leq 7$ (the rank of the $2 \times 2 \times 2$ tensor). And from this, you get an algorithm for MM.

Later we'll see this more generally — an upper bound on tensor rank of fixed-size matrix multiplication gives an upper bound on MM.

But here, we deviated from the optimal form Strassen said. There, you could take linear combinations of *both* the x and y variables, and multiply them; here we've restricted to only taking linear combinations of x and y.

Question 18.20. Do we lose something?

The answer is yes, but not much — it's just a constant factor.

Theorem 18.21

Over infinite fields, for every bilinear problem $\mathcal{F}$, we have $C^{*/}(F) \leq \text{rank}(\mathcal{F}) \leq 2C^{*/}(\mathcal{F})$.

So even though this sort of algorithm corresponding to the rank *can* be a bit worse than the optimal thing Strassen said, it can't be much worse — at most twice. So if we look at $n \times n$ matrix multiplication as n grows, the 2 is not going to matter, and without loss of generality we might as well just look at rank.

Proof. For the first part, any algorithm that looks lik ethe rank form is a special case of one that looks of Strassen's form. Rank says we can get ℓ products this way; because it's a special case you get ℓ products of that form, and Strassen's theorem says that's the best algorithm you can get.

So the only thing we ned to show is that if you get some expression using linear combinations of x and y's, you can turn it into one that only uses linear combinations of x's and combinations of y's, suffering just a factor of 2.

So let's do that. Suppose we have ℓ products

$$P_\lambda = \left(\sum_{i=1}^{n} u_{\lambda i} x_i + \sum_{j=1}^{m} v_{\lambda j} y_j\right)\left(\sum_{i=1}^{n} u'_{\lambda i} x_i + \sum_{j=1}^{m} v'_{\lambda j} y_j\right)$$

(so each factor is a linear combination of all the x's and y's). Having Ostrowski cost $C^{*/}(\mathcal{F}) = \ell$, by Strassen's theorem, means that you can get all your outputs as linear combinations of these things P_λ.

Now, we're just going to use distributivity; this becomes

$$\left(\sum_i u_{\lambda i} x_i\right)\left(\sum_j v'_{\lambda j} y_j\right) + \left(\sum_j v_{\lambda j} y_j\right)\left(\sum_i u'_{\lambda i} x_i\right) + \cdots.$$

There's some extras; but those extras are linear combinations of terms of the form $x_i x'_i$ and $y_j y'_j$ (these are the other two terms).

Now, this is all fine and good. But these yellow terms, we don't need them — because your output is a set of *bilinear* forms, you will never see terms of the form $x_i x_j$ or $y_i y_j$ — they must cancel somewhere in the final linear combinations. So we might as well not compute them.

So we're going to ignore that part. And then all we need to do is compute the blue and green things. We'll call the first thing P_1^λ; and we'll call the second thing P_2^λ, but flipped. Here we're using the fact that we have a field, so products are commutative — so we define

$$P''_\lambda = \left(\sum u'_\lambda x_i\right)\left(\sum v_{\lambda j} y_j\right).$$

So now we have 2ℓ products, instead of ℓ. And we know we can do linear combinations to get the outputs (and the yellow stuff $x_i x_j$ cancels, so they're not needed). So we get $\text{rank}(\mathcal{F}) \leq 2\ell = 2C^{*/}(\mathcal{F})$. □

Student Question. *We said we could switch because we're working over fields; but in Strassen's algorithm, you couldn't?*

Answer. Yes; in a ring you couldn't switch. But when we do the recursive algorithm, in the final layer you'll get a rank decomposition over a field; and we know that *final* thing is at most twice as bad. So even though in the middle you could have done something better... the only thing we know how to use recursively is something of this bilinear form, and in the *final* answer we're no worse than 2 off.

Remark 18.22. For 2×2 matrix multiplication, we actually know the rank is *exactly* 7. But for 4×4 and so on, it can actually drop — as n grows, it can get smaller and smaller. And this is why we'll define ω in the limit, because of the dropping.

There is a loss in going between rank and actual complexity — we'll see an example — but asymptotically it won't matter. This is Winograd's example.

Example 18.23

Suppose we have $M \times 2$ and $2 \times N$ matrices X and Y. For $M = 2$, this is 2×2 by $2 \times N$. We could multiply by Strassen's algorithm — if N is even, we can split into $N/2$ 2×2 MMs, and you'll get rank $\frac{N}{2} \cdot 7$ (if you did it with Strassen). This is the best-known rank upper bound for $(2 \times 2) \times (2 \times N)$.

But Winograd showed you can do better with a different algorithm that is not bilinear. He showed you can do $MN + M + N$ products; in this case, you get $2N + 2 + N = 3N + 2$, which is better than $\frac{7}{2}N$ (you can get a factor of 3 rather than 3.5, using an algorithm that is not bilinear).

Here is that algorithm: For every $i \in [M]$ and $j \in [N]$, we compute

$$P_{ij} \leftarrow (x_{i1} + y_{2j}) \cdot (x_{i2} + y_{1j}).$$

Then you do, for every $i \in [M]$, you compute $x_{i1} \cdot x_{i2}$; and for $j \in [N]$, you compute $y_{1j} \cdot y_{2j}$.

And now

$$z_{ij} = P_{ij} - x_{i1}x_{i2} - y_{1j}y_{2j}.$$

This is because P_{ij} expands out ot $x_{i1}x_{i2} + x_{i1}y_{1j} + x_{i2}y_{2j} + y_{2j}y_{1j}$; we care about the two middle terms, and the first two terms are subtracted off.

The number of products is MN in the first step, then M in the second, and N in the third. So you get $3M + 2$, which is a constant factor better than bilinear algorithms.

So you can actually gain by doing this. But if you have algorithms like this, you can't use them to recurse — they're not useful if you want recursive algorithms for bigger and bigger matrices, but they can be useful in terms of saving constant factors for smaller matrices.

As a recap, we took MM and cast it in this language of bilinear problems, a special case of quadratic problems. For quadratics we used Strassen to say the best algorithm is one that completely avoids divisions and has this special form. Then we defined the rank of a bilinear problem (or tensor) as corresponding to the best algorithm that looks like this, but with the x and y variables split up. Then we showed that only loses a factor of 2.

Next time we'll use bilinear decompositions to find recursive algorithms for MM. We'll define different notions of rank that are also useful.

§19 April 17, 2025

§19.1 Recap

Today we'll continue from where we were last time. Before that, we'll give a little recap of what we did, just so we're on the same page.

Last time, we talked about bilinear computational problems. These are problems on two sets of variables $x_1, \ldots, x_n$ and $y_1, \ldots, y_m$, and you have outputs $f_1, \ldots, f_K$, where

$$f_k = \sum_{i=1}^{n}\sum_{j=1}^{m} t_{ijk}x_iy_j.$$

This tensor (t_{ijk}) defines your bilinear problem.

> **Question 19.1.** What's the fastest algorithm for computing such a thing, restricted to the arithmetic circuit or SLP model?

Today, our field will be $\mathbb{F} = \mathbb{Q}$ (or you could use $\mathbb{R}$ or $\mathbb{C}$ if you want).

Last time, we discussed what these circuits have to look like; Strassen said that for any quadratic form, you can avoid divisions. In particular, your algorithms look like taking a bunch of linear combinations of the x and y variables, taking a bunch of these products, and then you can express your outputs as linear combinations of these products.

Then we restricted ourselves to *bilinear* algorithms, which correspond to rank decompositions of your tensor. Those look like this: Let's say you have r vectors $u_\lambda \in \mathbb{F}^n$, $v_\lambda \in \mathbb{F}^m$, and $w_\lambda \in \mathbb{F}^K$ (for $\lambda = 1, \ldots, r$). For each of these λ's, you have

$$P_\lambda = \left(\sum_i u_{\lambda i}x_i\right) \cdot \left(\sum_j v_{\lambda j}y_j\right).$$

And then your outputs are

$$f_k = \sum_{\lambda=1}^{k} w_{\lambda k} P_\lambda.$$

So you take linear combinations of the x-variables and y-variables and take products, and then each of your outputs is a linear combination of these products.

Definition 19.2. The minimum r such that this exists is the rank of the tensor t.

You can express this as

$$t = \sum_{\lambda=1}^{r} u_\lambda \otimes v_\lambda \otimes w_\lambda,$$

where this is a rank-1 tensor whose (i,j,k)th coordinate is $u_{\lambda i} v_{\lambda j} w_{\lambda k}$.

And we showed last time that the rank here is within a factor of 2 of the best number of products in an arbitrary algorithm in the arithmetic circuit model for this sort of problem — so if we only count products and divisions, then the rank is what we need to study.

Definition 19.3. We define $C(t)$ as the minimum size of a SLP computing the bilinear problem that t defines. We define $C^*(t)$ as the minimum number of products and divisions of an SLP computing this bilinear problem.

So we know that $C^*(t)$ is within a factor of 2 of the rank — we have that

$$C^*(t) \leq R(t) \leq 2C^*(t).$$

§19.2 Three definitions of ω

Now we can define three different ω's, and we'll show they're the same. The first is the one we actually care about — the exponent of the runtime of matrix multiplication in an arithmetic circuit model.

Definition 19.4. We define $\langle n, n, n\rangle$ as the $n \times n$ matrix multiplication tensor.

Recall that it has the following entries — for $i, j, k, i', j', k' \in [n]$, we have

$$\langle n, n, n\rangle_{(i,j'),(j,k'),(k,i')} = \delta_{ii'}\delta_{jj'}\delta_{kk'},$$

where δ_{ab} is 1 if $a = b$ and 0 otherwise.

Definition 19.5. We define $\omega = \inf\{\rho \mid C(\langle n, n, n\rangle) \leq O(n^\rho)\}$.

So this is really what we care about. The size of the circuit involves *all* operations — additions, multiplications, subtractions, operations with scalars, and so on.

Definition 19.6. We define $\overline{\omega} = \inf\{\rho \mid R(\langle n, n, n\rangle) \leq O(n^\rho)\}$.

This one is just the rank — it doesn't care about additions or scalar multiplications.

Definition 19.7. We define $\overline{\omega}' = \inf\{\rho \mid C^{*/}(\langle n, n, n\rangle) \leq O(n^\rho)\}$.

This is just the number of products and divisions you need to do. By what we showed last time — that $C^{*/}(t) \leq R(t) \leq 2C^{*/}(t)$ — we know that $\overline{\omega} = \overline{\omega}'$.

The second definition also completely ignores additions and scalar operations and so on, so we also know $\overline{\omega} \leq \omega$ (since the first definition counts strictly more things). We're going to prove that the opposite is also true.

Theorem 19.8

We have $\omega \leq \overline{\omega}$.

This means $\omega = \overline{\omega}$. So to study the runtime of MM, we only need to study the rank of the tensor — we don't care about additions or scalar operations.

The way the proof works will basically be what we did for Strassen's algorithm — we'll take a base algorithm for some constant-sized matrix multiplication, and run recursion. And we'll show additions and scalar operations don't matter — they'll be subsumed in the $O(\bullet)$.

We'll prove this, and then we'll look at various operations on tensors — permuting indices and slices and so on. This will allow us to look at rectangular MM of different dimensions, and show any permutation of the dimensions leaves the rank the same.

Proof. The definition of $\overline{\omega}$ means that for every ε, there exists some constant m_0 such that for every $m > m_0$, we have $R(\langle m, m, m\rangle) \leq m^{\overline{\omega}+\varepsilon}$. (There's an $O(\bullet)$, which involves some constant, but we can put that constant into the m^{ε}). Fix some $\varepsilon > 0$, and fix some m for which this holds. Now we're going to look at an algorithm for multiplying $n \times n$ matrices given this.

This is only a rank expression; it has this many products, but it also has a bunch of additions and scalar operations. Let ℓ be the number of additions and scalar operations in the rank decomposition that gives us this bound. If m is constant, then ℓ is some constant in terms of m (it could be really big, but we don't care).

Now we're going to take an $n \times n$ matrix multiplication problem, where n is some power of m. (If n is not a power of m, you can grow it a bit to a power of m; it'll only grow by a factor of m, so that doesn't matter.)

Now we'll do exactly what we did with Strassen's algorithm. We have an $m^i \times m^i$ input X and another one Y, and XY is supposed to equal Z.

We split each into buckets, now each looks like an $m \times m$ matrix multiplication, except that the entries are $m^{i-1} \times m^{i-1}$ matrices.

And now we follow our algorithm in the rank decomposition, which says we take a bunch of linear combinations of the X matrix (on those blocks), and similarly with the Y matrix. So you take your $\sum_i u_{\lambda i} x_i$ combinations, and similarly with the Y matrix. Then we take r products. And in the output, we again take linear combinations.

We said the total number of additions and scalar operations on these blocks is ℓ.

So because of this, we get to write an expression. Let $A(i)$ be the size of the circuit for $m^i \times m^i$ matrix multiplication. And now we get to write this as a recurrence. We have to do r recurrences, which correspond to matrix products of size $m^{i-1} \times m^{i-1}$ (just like in Strassen, where we had $n/2 \times n/2$). And then we also have ℓ number of additions and scalar operations on matrices of size $m^{i-1} \times m^{i-1}$. Each of these additions and scalar operations take at most linear time in the size of the matrix (because addition is pointwise). So we get

$$A(i) \leq r \cdot A(i-1) + \ell \cdot m^{2(i-1)}.$$

(The $m^{2(i-1)}$ is the time to add two matrices or multiply a matrix by a scalar.)

Now we have to solve this recurrence. The base case $A(0)$ is for multiplying 1×1 matrices; that's just 1.

Now let's write this all out. We'll think about this with a recursion tree approach. So we have a tree with an expansion of r; and in the beginning we have m^i, after we recurse now we have matrices of dimension m^{i-1}, then we have r^2 matrices of size m^{i-2}, and so on; in general, we have r^j of size m^{i-j}, and at the bottom we have r^i of size 1. (At the very bottom of the recursion tree, we have r^i MM problems, but their size is 1×1.)

So then we get to expand this out as

$$A(i) = r^i \cdot 1 + \sum_{j=0}^{i-1} r^j \cdot \ell \cdot m^{2(i-j-1)}.$$

Moving things around, we can write this as

$$r^i + \ell \cdot m^{2(i-1)} \cdot \sum_{j=0}^{i-1} \left(\frac{r}{m^2}\right)^j.$$

Now, we know that $r = m^{\overline{\omega}+\varepsilon}$. And notably, we know $\overline{\omega} \geq 2$. So $r > m^2$. That means this thing r/m^2 is bigger than 1. And

$$\sum_{j=0}^{i-1} z^j = \frac{z^i - 1}{z - 1}.$$

If we apply that here, we get

$$r^i + \ell m^{2(i-1)} \cdot \frac{\left(\frac{r}{m^2}\right)^i - 1}{(r/m^2) - 1}.$$

We'll drop the -1 in the numerator, because it's pesky; and then we get

$$r^i \left(1 + \frac{\ell m^{2(i-1)}}{(r - m^2) \cdot m^{2(i-1)}}\right).$$

Notably, the $m^{2(i-1)}$s cancel, so this equals

$$r^i \left(1 + \frac{\ell}{r - m^2}\right).$$

And the secodn part is just some constant in terms of m (for any constant m, this thing is a constant).

This means for every constant $\varepsilon > 0$, there exists a constant m such that

$$C(\langle m^i, m^i, m^i \rangle) \leq O_m(r^i) \leq O_m((m^{\overline{\omega}+\varepsilon})^i)$$

for every i. Notably, thsi means

$$C(\langle n, n, n \rangle) \leq O_m(n^{\overline{\omega}+\varepsilon}).$$

□

Student Question. *We said that $\ell = O(1)$; where does that come from?*

Answer. We're promised a rank decomposition for $m \times m \times m$ matrix multiplication where m is a *constant*. So you have a rank decomposition; and ℓ is some constant in terms of m that comes from this decomposition.

So this shows $\omega \leq \overline{\omega}$. The point is additions and scalar multiplications and so on get completely subsumed in the $O(\bullet)$.

Student Question. *In the definition of ω, we used* inf*; does this mean* log *factors disappear?*

Answer. Yes. It also turns out you need the inf — if you have a bound on the rank for some constant-sized thing, it's a theorem that you can get a *better* bound for bigger things. So it's limiting; but the limit exists.

So now we're just going to study rank, and we're going to completely forget about additions — additions are cheap, so they can be subsumed in the $O(\bullet)$.

This is basically the same as Strassen, but for arbitrary n and given bound.

Remark 19.9. It also means that if for some fixed m you prove that $R(\langle m, m, m\rangle) \leq r$, then $\omega \leq \log_m r$ (just like having a bound of 7 for 2×2 gave $\log_2 7$).

§19.3 Permutations of tensors

Now we're going to look at rectangular MM, and more generally at what you do when you permute things in the tensor.

We'll see what permutations mean in different representations of a tensor.

§19.3.1 Permutation of variables

The first thing we're going to do is permute the variables. First we'll draw a picture of what that means; then we'll look at the trilinear representation and see the same thing, and then we'll talk about the tensor itself and see the same thing.

Imagine you have a tensor, which is some box in 3D space (with axes corresponding to x, y, and z). The permutation of the variables means, say I decide to permute x and y. Now x oes on the vertical axis and y on the horizontal, but z stays the same (it's still into the page). It's the same tensor, except that I rotated it.

In tems of the trilinear representation, your original tensor looks like

$$\sum t_{ijk} x_i y_j z_k$$

(where $i \in [n]$, $j \in [n]$, and $k \in [K]$). When I decide to permute x and y, I get a new tensor that looks like

$$\sum t_{ijk} y_j x_i z_k.$$

More generally, I can take any permutation $\pi \in S_3$, and I can apply it to the x, y, and z locations; and this will give me a new tensor πt. So t used to lie in $\mathbb{F}^m \times \mathbb{F}^n \times \mathbb{F}^K$, and now we get a new tensor πt where we flipped where they are.

To make notation easier, we'll say $t \in \mathbb{F}^{n_1 \times n_2 \times n_3}$, so $m = n_1$, $n = n_2$, and $K = n_3$. Then

$$\pi t \in \mathbb{F}^{n_{\pi(1)} \times n_{\pi(2)} \times n_{\pi(3)}}$$

is given by

$$(t\pi)_{i_{\pi(1)}} i_{|pi(2)} i_{\pi(3)} = t_{i_1 i_2 i_3}.$$

So it's the same tensor, but I flipped where things go, based on the permutation.

What would happen to the rank of a tensor when I flip around variables like this? It has to stay the same — you take any rank decomposition and reorder things, and you get the same thing.

Claim 19.10 — For every $\pi \in S_3$, we have $R(\pi t) = R(t)$.

The proof is just reordering the terms of the rank decomposition.

Now we're going to look at the MM tensor, and we're going to permute it and see what happens. Let's say we want to look at the MM tensor for $\langle M, N, K\rangle$ (this is an $M \times N$ matrix times a $N \times K$ matrix; so you output hte transpose of the product, which is a $K \times N$ matrix). This is given by

$$\langle M, N, K\rangle_{ij',jk',ki'} = \delta_{ii'}\delta_{jj'}\delta_{kk'}$$

where $i, i' \in [M]$, $jj' \in [N]$, and $kk' \in [K]$.

now suppose I apply the permutation $\pi = (123)$ (in cycle notation). Then in

$$(\pi\langle M, N, K\rangle)_{ki',ij',jk'} = \langle M, N, K\rangle_{ij',jk'ki'} = \delta_{ii'}\delta_{jj'}\delta_{kk'}.$$

What does this mea? This is $\langle K, M, N\rangle$ (exactly by definition — I'v erotated the indices).

And now, by the obvious claim, we immediately get that

$$R(\langle M, N, K\rangle) = R(\langle K, M, N\rangle) = R(\langle N, K, M\rangle).$$

So rotating the dimensions really doesn't matter in terms of the rank.

This is only for 3 permutations. Now, for the next thing, we're goign to define a different permutation of tensors. There's 6 of these things, but we'll show that for any of them, the rank stays the same (this one only covers 3).

§19.3.2 Permutation of slices

This first part was permutation of variables; the next is permutation of slices.

What's a slice? Let's look at our tensor. Our x variables are on the horizontal axis. I could decide to permute the x variables however we want; that takes any of these slices and flips them around. I could also do this with y or z.

Example 19.11

We could take $t \in \mathbb{F}^{M\times N\times K}$ and take $\sigma \in S_M$. If we apply this to the x slices, then we get a new tensor t' given by

$$t'_{\sigma(i)jk} = t_{ijk}.$$

We can do the same with the y and$z$'s.

Clearly, just permuting the order of the x-variables, I can apply this perutatio t to rank decomposition, and the rak will stay the same.

Claim 19.12 — Rank is invariant under permutations of slices.

So agaim I just reorder things, but within each slice separately. Rewriting what we had before, which was

$$\langle M, N, K\rangle_{ij',jk',ki'} = \delta_{ii'}\delta_{jj'}\delta_{kk'}$$

for $i, i' \in [M]$ and so on —- the first thing we'll do is apply a permutation of the variables $\pi = (12)(3)$. Then we have

$$(\pi\langle M, N, K\rangle)_{jk',ij',ki'} = \delta_{ii'}\delta_{jj'}\delta_{kk'}.$$

Now we permute the x-slice by $jk \to k'j$ and the y slice by $ij' \to j'i$ —- so I flip these things within each slice. We also flip the z-slice.THen we get

$$t_{k'j,j'i,i'k} = \delta_{ii'}\delta_{jj'}\delta_{-kk'}.$$

And this is exactly the $\langle K, , N, M\rangle$ matrix multiplicatio tensor.

So we used to be $\langle M, N, K\rangle$, and then we get $\langle K, N, M\rangle$. THis means I rotated the first and third.

Now I've gotten $R(\langle M, N, K\rangle) = R(\langle K, M, N\rangle) = R(\langle N, K, M\rangle)$; and I've also shown these are equal to $R(\langle K, N, M\rangle)$. And I can again apply the permutation (123) to get that this one is also equal to $R(\langle M, K, N\rangle) = R(\langle N, M, K\rangle)$. And now I get all the permutations.

What we've gotten from this is that becuase the MM tensor is so symmetric, any permutations of dimensions for rectangular MM has the same rank.

Why is this useful? If we care about square multiplication tensors, it turns out to be useful because we only need to bound the rank of *rectangular* MM, and we'll be able to get a bound on the square. To prove this, we'll need to define some more things; but that's the goal. Before we were just restricted to square ones, but now we can go to rectangular ones.

§19.4 Direct sum of tensors

What we're going to do next is, we're going to define direct sums and products of tensors. Given two tensors, you can define a new tensor which is their sum, and one which is their product. The product is useful because it basically corresponds to recursion in our algorithmic world. The sum is useful because sometimes it means you're computing two independent copies of the same problem, and it turns out sometimes that can be more efficient than computing them separately, which is surprising.

Let's say we have a tensor $t \in \mathbb{F}^{K\times M\times N}$, and another tensor $t' \in \mathbb{F}^{K'\times M'\times N'}$. (We draw t and t' as boxes.)

The direct sum of t and t' is to take t and t' and put them in the same space, and put 0's everywhere else (and they don't overlap). We draw them as two boxes touching at their corner.

So it's basically a new tensor saying, 'take these two things and compute them together.'

Definition 19.13. We define $t \oplus t' \in \mathbb{F}^{(K+K')\times(M+M')\times(N+N')}$ by

$$(t \oplus t')_{ijk} = \begin{cases} t_{ijk} & i \le K, j \le M, k \le N \\ t'_{(i-K)(j-M)(k-N)} & i > K, j > M, k > N \\ 0 & \text{otherwise.} \end{cases}$$

Because of the way it's written, what do you think is the rank of this thing?

Claim 19.14 — We have $R(t \oplus t') \le R(t) + R(t')$.

This is because you have two rank decompositions, one for t and one for the other; you just extend the vectors defining those rank decompositions by tacking on 0's after in the one for t and before in the one for t'.

So this is true.

Conjecture 19.15 (Strassen) **—** We always have $R(t \oplus t') = R(t) + R(t')$.

So he conjectured that it should always be equal. But despite many attempts to prove it, someone actually disproved it.

Theorem 19.16 (Shitov 2019)
No! There exist t and t' such that $R(t \oplus t') < R(t) + R(t')$.

These t and t' are very large. Then there was also a paper in 2019 that said if your tensors are actually small (e.g., the rank of one of them is at most 6), then actually the conjecture is true. So for small tensors it turns out to be true, but in general it's not true.

Actually, the fact that it can be strict is good for algorithms — if you can somehow take two copies of the matrix multiplication tensor and compute them faster than individually computing them, then it's actually very good algorithmically. And we will use something like this later (not exactly this, but something else).

§19.5 Kronecker product

That's the direct sum; the next thing we'll do is the product.

You have your two tensors $t \in \mathbb{F}^{K \times M \times N}$ and $t' \in \mathbb{F}^{K' \times M' \times N'}$; so t has coordinates t_{ijk} where $i \in [K]$, $j \in [M]$, and $k \in [N]$, and likewise with t'. This one is going to have *pairs* of indices.

Definition 19.17. The Kronecker product is defined as

$$(t \otimes t')_{ii',jj',kk'} = t_{ijk} \cdot t'_{i'j'k'}.$$

So as a picture, imagine we draw one tensor as a $2 \times 2 \times 2$ cube, and another as a $2 \times 1 \times 1$ cube. Then you take the first tensor, and think about it as kind of blown up; and within each of the cubes, you put a copy of the second tensor. (We draw those copies in yellow.)

So the indices used to be (i, j, k), but now there's pairs — there's an i for the big tensor and an i' for the tensor you put inside it, and so on; and those tell you which of these little pieces to go in.

Student Question. *Does this tensor have* 6 *dimensions, or is it still* 3*-dimensional?*

Answer. It's still 3-dimensional, where ii' is the first index, jj' is the second, and kk' is the third — the indices are just pairs now, rather than numbers.

Claim 19.18 — We have $R(t \otimes t') \leq R(t) \cdot R(t')$.

And there are tensors — in particular, 2×2 matrix multiplication — where this can be strict.

Proof. How you do this is not hard. You take the rank decomposition of t and of t', so say

$$t = \sum_{\lambda=1}^{r} a_\lambda \otimes b_\lambda \otimes c_\lambda \quad \text{and} \quad t' = \sum_{\mu=1}^{s} f_\mu \otimes g_\mu \otimes h_\mu$$

(where $a_\lambda \in \mathbb{F}^K$, $b_\lambda \in \mathbb{F}^M$, $c_\lambda \in \mathbb{F}^N$, $f_\mu \in \mathbb{F}^{K'}$, $g_\mu \in \mathbb{F}^{M'}$, $h_\mu \in \mathbb{F}^{N'}$). So you have these vectors that define the rank decomposition. Now what you do to get a rank decomposition of these guys is, you're just going to define some new vectors $A_{\lambda,\mu} \in \mathbb{F}^{KK'}$, $B_{\lambda,\mu} \in \mathbb{F}^{MM'}$, $C_{\lambda,\mu} \in \mathbb{F}^{NN'}$ in the following way: We just define

$$(A_{\lambda\mu})_{ii'} = a_{\lambda i} f_{\mu i'},$$

and similarly $(B_{\lambda\mu})_{jj'} = b_{\lambda j} g_{uj'}$, and so on. So you're just kind of multiplying the corresponding vectors pointwise. And $(C_{\lambda\mu})_{kk'} = c_{\lambda k} h_{\mu k'}$.

There's rs of these, and you can check that $t \otimes t' = \sum_{\lambda,\mu} A_{\lambda\mu} \otimes B_{\lambda\mu} \otimes C_{\lambda\mu}$, just by looking at the coordinates and the way things are defined. So you basically take the tensor product of these vectors and get longer vectors, and they define your tensor decomposition. □

However, this is not an equality — it can be a strict inequality, and this is actually what helps us get better bounds on ω than $\log_2 7$.

§19.5.1 Matrix multiplication and the Kronecker product

Why do we care about the Kronecker product? Because of MM — the Kronecker product of two MM tensors is a MM tensor itself. That's because if you look at the above picture, it exactly corresponds to the recursive nature of MM — you can take a bigger matrix and split it up into blocks. And in a MM problem, the problem of multiplying the blocks inside of it is still a MM problem. This exactly corresponds to the fact that MM is just, if you multiply it by itself, you get another MM tensor of a bigger size. We'll write that down, but that's basically why we care.

Let's say I have a MM tensor $\langle M, N, K\rangle_{ij'jk'ki'} = \delta_{ii'}\delta_{jj'}\delta_{kk'}$, and I also have another one $\langle M', N', K'\rangle_{pq'qr'rp'} = \delta_{pp'}\delta_{qq'}\delta_{rr'}$, where $p, p' \in [M']$ and so on.

I take these two. They have different dimensions, and they're defined like this. And now I'm going to look at their Kronecker product

$$\langle M, N, K\rangle \otimes \langle M', N', K'\rangle.$$

The Kronecker product is defined on pairs of indices. So we'll have ij' from the first thing, and then pq' from the second; and that defines our first index. And so on. So the coordinates of this look like

$$\langle M, N, K\rangle \otimes \langle M', N', K'\rangle_{ij'pq',jk'qr',ki'rp'}.$$

And by definition, this is just the product of the two thigns — so this is just

$$\delta_{ii'}\delta_{jj'}\delta_{kk'}\delta_{pp'}\delta_{qq'}\delta_{rr'}.$$

But that's exactly the same as — $\delta_{ii'}$ is 1 whenever $i = i'$, and $\delta_{pp'}$ is 1 whenever $p = p'$. So I can write

$$\delta_{ii'}\delta_{pp'} = \delta_{(i,p),(i',p')}$$

(now I'm checking equality between *pairs* of indices). And we can do the same with the others — we can write

$$\delta_{(j,q),(j',q')} \quad \text{and} \quad \delta_{(k,r)(k',r')}.$$

So I pair i with p and j with q, i with p' and i' with p', and so on; and then this becomes isomorphic to the MM tensor where the indices are (i,p), (i',p'), (j,q), (j',q'), (k,r), (k',r'). So this is the same thing as

$$\langle MM', NN', KK'\rangle_{ip,j'q',jq,k'r',kr,i'p'}.$$

So the Kronecker product of two MM tensors is just the MM tensor where you multiply the dimensions.

And this exacty corresponds to the following: You have an $MM' \times NN' \times KK'$ MM, and you block the X matrix and the Y matrix and the Z matrix into $M \times N \times K$ blocks, where each one inside is an $M' \times N' \times K'$ — this is exactly the same as what we did with the algorithm.

So this Kronecker product means I can multiply an $MM' \times NN'$ matrix by an $NN' \times KK'$ matrix, by splitting into blocks of dimensions $M' \times N'$ and $N' \times K'$. That's what this means — so the Kronecker product (for MM) corresponds exactly to recursion.

Now we can prove the final lemma of today's lecture. This is that you can use rectangular matrix multiplication to get a bound on ω.

Lemma 19.19

If $R(\langle K, M, N\rangle) \leq r$, then $\omega \leq 3\log(r)/\log(KMN)$.

So there's some constant matrix multiplication tensor of dimensions $K \times M$ and $M \times N$, and you somehow manage to figure out that its rank is at most r. And then you get this bound. So for any kind of dimensions of a rectangular matrix, if you bound the rank, then you immediately get a bound on ω.

Now we'll prove this.

Proof. Consider $\langle KMN, KMN, KMN\rangle$. This is the matrix multiplication tensor where the dimensions are all KMN. This one, we know, is isomorphic to

$$\langle K, M, N\rangle \otimes \langle N, K, M\rangle \otimes \langle M, NK\rangle.$$

(This is because when I multiply MM tensors, I just multiply the dimensions, as seen above.)

Now, we know that $R(\langle KMN, KMN, KMN\rangle)$ is at most the product of the ranks. But we also know the rank doesn't change when you permute the dimensions. So we get that this is at most $R(\langle K, M, N\rangle)^3 = r^3$.

Now we use the claim from before that when you have a bound on the MM tensor rank for *square* MM, we get a bound on ω. So from this, we get that

$$\omega \leq \log_{KMN} r^3 = \frac{3\log r}{\log KMN}.$$ □

If you think about it in the recursive algorithmic world, what you do is take a MM problem like $\langle KMN, KMN, KMN\rangle$ block it into $\langle K, M, N\rangle$ subtensors which look like $\langle N, K, M\rangle \otimes \langle M, N, K\rangle$, and block those. So there's 3 recursive levels where you're doing different things; and in each of those you do a r rank decomposition.

Remark 19.20. In the notes, Virginia has written down all the small rectangular matrix multiplication tensors for which we know a bound on the rank. Basically, the one that gives you the best bound is by Pan, and he showed that

$$R(\langle 70, 70, 70\rangle) \leq 143640,$$

which gives that $\omega < 2.8$ over any field. Recently, there was a big splash of deep learning and Alpha Tensor and deriving a new MM algorithm 'beating Strassen.' It doesn't beat Strassen, it only beats Strassen over $\mathbb{GF}_2$. What it did was show that

$$R(\langle 4, 4, 4\rangle) \leq 47$$

over GF(2). You can prove that their decomposition can't be lifted to $\mathbb{R}$ or $\mathbb{C}$. The best before was two iterations of Strassen, which would give you $7^2 = 49$. But this is still the best for general fields.

If you use $R(\langle 4, 4, 4\rangle) \leq 47$, you can get a practical MM algorithm over GF(2) with $\omega \approx 2.778$; it only works over GF(2), but could be more practical than Pan's (which involves large matrices).

So this is basically the best we can do with the rectangular matrix approach.

Student Question. *What is GF(2)?*

Answer. Any field where the characteristic is 2.

§20 April 22, 2025 — Border rank, Bini et. al., introduction to Schönhage's theorem

Today we're going to look at an extension of rank called *border rank*. And then we'll see an example border rank decomposition which will allow us to get a much better bound on ω than we could've gotten just with rank techniques. Then we'll discuss but not prove Schönhage's theorem (we'll prove it next time).

§20.1 Tensor rank in limits

The first thing Virginia will say is that order-3 tensors are weird. There's actually a very big difference between matrices and tensors when it comes to rank.

For matrices, let's imagine we have a sequence of matrices A_1, A_2, …, A_j, where in the limit as $j \to \infty$, you get some matrix A. And suppose I promise you that for every one of these matrices, its rank is bounded by r. Then it turns out we can say the rank of A is *also* bounded by r.

Fact 20.1 — If $\lim_{j\to\infty} A_j = A$ and $\text{rank}(A_j) \leq r$ for all j, then $\text{rank}(A) \leq r$.

So if the rank of every element in the sequence is at most r, then this is also true in the limit.

This is roughly because for a matrix to have rank r just means that all the $(r+1)\times(r+1)$ minors vanish (i.e., the sub-determinants of size $(r+1)\times(r+1)$ are zero) for every A_j. And for matrices, these sub-determinants are continuous functions. So if they vanish everywhere in the sequence, then in the limit they'll also vanish. That means the $(r+1)\times(r+1)$ minors of A also vanish, which is the same as saying $\text{rank}(A) \leq r$.

So this is true for matrices. What's very strange is that this is *not* true for order-3 tensors — you can have a tensor A whose rank is r, but you can have a sequence of tensors which go to A but which all have rank less than r. So you can have a sequence of tensors going to the one you care about, whose rank is strictly smaller than that of the tensor.

This turns out to actually be useful for designing algorithms! That's because you can use these sequences going to the tensor you care about, and their rank is smaller than what you'd have gotten using the tensor themselves. It turns out you can turn this into an algorithm, and that's what this lecture is about.

Let's see an example.

Example 20.2

Consider the tensor $t = a_0b_0c_0 + a_1b_0c_1 + a_0b_1c_1$ in trilinear notation.

It's very easy to see that this has rank at most 3 (because there are three products). It's also known this tensor has rank *exactly* 3. But we'll see a sequence of tensors whose rank is 2 that approach this.

Let $\varepsilon > 0$ (this is some value). We'll define a tensor $t(\varepsilon)$ by giving a rank-2 decomposition of it — we define

$$t(\varepsilon) = (1,\varepsilon)\otimes(1,\varepsilon)\otimes(0,1/\varepsilon) + (1,0)\otimes(1,0)\otimes(1,-1/\varepsilon).$$

There are two terms, so this is a rank-2 tensor — we have $R(t(\varepsilon)) = 2$ for every ε, while $R(t) = 3$.

Now all we have to do is prove that as $\varepsilon \to 0$, we have $t(\varepsilon) \to t$. (If you're not used to this triad notation, we'll expand it in trilinear form so you can see what it looks like.)

Claim 20.3 — We have $\lim_{\varepsilon\to 0} t(\varepsilon) = t$.

Proof. We'll expand what $t(\varepsilon)$ means in trilinear notation. Recall that in something like $(1,\varepsilon)\otimes(1,\varepsilon)\otimes(0,1/\varepsilon)$, these numbers correspond to coefficients in terms of a_0 and a_1, b_0 and b_1, c_0 and c_1, respectively. So we can expand this as

$$t(\varepsilon) = (a_0+\varepsilon a_1)\cdot(b_0+\varepsilon b_1)\cdot\frac{c_1}{\varepsilon} + a_0\cdot b_0\cdot\left(c_0-\frac{c_1}{\varepsilon}\right).$$

Now we can just spread some stuff around and use distributivity; this becomes

$$\frac{a_0b_0c_1}{\varepsilon} + a_0\varepsilon b_1\frac{c_1}{\varepsilon} + \frac{\varepsilon a_1b_0c_1}{\varepsilon} + \frac{\varepsilon^2a_1b_1c_1}{\varepsilon} + a_0b_0c_0 - \frac{a_0b_0c_1}{\varepsilon}.$$

Some of these cancel, and this becomes

$$a_0b_1c_1 + a_1b_0c_1 + a_0b_0c_1 + \varepsilon a_1b_1c_1.$$

The first three terms are our tensor t, so we've shown that $t(\varepsilon) = t + \varepsilon\cdot a_1b_1c_1$. As $\varepsilon\to 0$, the extra term disappears, so what we get is t. □

So in the limit as $\varepsilon\to 0$, this rank-2 tensor family goes to t. So we have a sequence of tensors whose rank is 2, but they go in the limit to a tensor whose rank is 3. So what we had for matrices does *not* hold for tensors.

But more interestingly, you can think of this sequence of tensors as an approximation to t — for each ε we have something getting closer and closer to t. This was used to get an *approximation* algorithm for matrix multiplication, but it turns out you can use it to get an *exact* algorithm as well!

§20.2 Border rank

This was a nice motivating example; now we're going to define what it's doing.

We have some field $\mathbb{F}$, just like last time. We're actually going to talk about $\mathbb{F}[\varepsilon]$, the ring of polynomials in ε with coefficients from your field — i.e., the ring of polynomials of the form $\sum_{i=0}^{m} a_i\varepsilon^i$ (for $a_i\in\mathbb{F}$).

What we're going to do is look at rank decompositions where your coefficients in the linear combination are polynomials in ε, instead of coefficients from the field.

So we'll let $t\in\mathbb{F}^{K\times M\times N}$ be some tensor we care about (with K x-variables, M y-variables, and N z-variables), and let $h\geq 0$ be an integer.

Definition 20.4. We define the h-rank of t, denoted $R_h(t)$, as the smallest r such that there exist r triples of vectors $u_\ell\in\mathbb{F}[\varepsilon]^K$, $v_\ell\in\mathbb{F}[\varepsilon]^M$, and $w_\ell\in\mathbb{F}[\varepsilon]^N$ (f0r $\ell=1,\ldots,r$) such that

$$\sum_{\ell=1}^{r} u_\ell\otimes v_\ell\otimes w_\ell = \varepsilon^h\cdot t + O(\varepsilon^{h+1}).$$

So for each $\ell=1,\ldots,r$, we have vectors u_ℓ, v_ℓ, and w_ℓ of lengths K, M, and N whose entries are *polynomials* in ε (over your field).

The tensor $t\in\mathbb{F}^{K\times M\times N}$ doesn't have any ε's — it's just a normal tensor over your field. And we're saying there's r of these vectors which have entries which are polynomials in ε, and we look at the sum of their triads. We're saying that if we collect all the terms with a coefficient of ε^h, then we get t exactly. There are no summands whose degree in ε is less than h. And the rest of terms have a degree in ε which is strictly bigger than h — the $O(\varepsilon^{h+1})$ term collects all terms with ε^j for $j>h$.

We'll rewrite the earlier example in terms of this in a second. But we have a bunch of vectors with polynomials in ε; when you multiply them out and take the sum, you get something where your tensor appears with a coefficient of exactly ε^h, and then you have some stuff with ε^{h+1} or more.

In our example, our decomposition doesn't have polynomials in ε (instead we have $1/\varepsilon$). But we can turn it into a polynomial by multiplying by ε — we multiply both parts by ε to get

$$\varepsilon t(\varepsilon) = (1,\varepsilon)\otimes(1,\varepsilon)\otimes(0,1) + (1,0)\otimes(1,0)\otimes(\varepsilon,-1).$$

Then when we expand things out, both things will get multiplied by ε; so we'll get that

$$\varepsilon t(\varepsilon) = \varepsilon t + \varepsilon^2 \cdot a_1 b_1 c_1.$$

These are all polynomials in ε (with degree at most 1). And what we get here is that if we say $\varepsilon t(\varepsilon) = t'(\varepsilon)$, then we get that

$$R_1(t) \le 2,$$

because the degree here is 1.

So I got some decomposition where none of the terms have degree in ε less than 1; and when I collect the terms that have exactly a coefficient of ε I get the tensor I care about, and the rest of the terms are some error stuff.

Intuitively, what's happening here is that I have some tensor $t(\varepsilon) = \varepsilon^h \cdot t + O(\varepsilon^{h+1})$. And intuitively, what's happening is that $(1/\varepsilon^h)\cdot t(\varepsilon) \to t$ as $\varepsilon \to 0$ (if I divide both sides by ε^h) — except that I want to work with polynomials, so we write things in the first kind of expression.

So the h-rank is the smallest r such that you can write $\varepsilon^h \cdot t$ plus some error term as a rank decomposition with coefficients which are polynomials in ε.

Student Question. *How do we relate the exponent of ε to the rank of t?*

Answer. There's no known relationship. If you can find some h such that you can get an expression like this with a small number of terms, then you're showing the h-rank is at most r.

But there are techniques where for some tensors, you can show that no matter how big h you pick, you can't beat a certain rank.

Student Question. *In this case, how do we know the rank is 2?*

Answer. We defined $t(\varepsilon)$ by giving a rank-2 decomposition of it (and you can show you can't do better, because there's two variables in each part and the tensor depends on each of these pairs).

Virginia has a question for us: Can we put a bound on the degree of the polynomials in ε appearing in u_ℓ, v_ℓ, and w_ℓ (given that h is fixed)? Do I need any terms with degree more than h? No — if we have anything with degree more than h, we can just get rid of them. Those terms would go into the $O(\varepsilon^{h+1})$ error anyways, so we might as well get rid of it.

So each thing in u_ℓ, v_ℓ, and w_ℓ is a polynomial of degree at most h. This means if $h = 0$, we're looking at rank decompositions where the ε-degree is 0, which is exactly the rank.

Also, we know $R_0(t) \ge R_1(t)$, since any rank decomposition with degrees at most 0 is also one with degrees at most 1. So we get the following observation:

Claim 20.5 — We have $R(t) = R_0(t) \ge R_1(t) \ge \cdots \ge R_h(t) \ge R_{h+1}(t) \ge \cdots$.

So we define the *border rank* as basically the smallest rank that you can achieve with any h.

Definition 20.6. The border rank of t is defined as $\underline{R}(t) = \min_{h \geq 0} R_h(t)$.

This is kind of weird, but it's actually useful. First, here's a lemma we're going to use.

Lemma 20.7

Suppose we have two tensors $t \in \mathbb{F}^{K \times M \times N}$ and $t' \in \mathbb{F}^{K' \times M' \times N'}$. Then the following are true:

(1) For every $h \geq 0$, if t' is a permutation of t (either in variables or slices), then $R_h(t) = R_h(t')$.

(2) For every h and h', we have

$$R_{\max(h,h')}(t \oplus t') \leq R_h(t) + R_{h'}(t').$$

(3) For every h and h', we have

$$R_{h+h'}(t \otimes t') \leq R_h(t) \cdot R_{h'}(t').$$

For (1), last class, we saw two notions of permutations — either we can permute the variables (where x becomes y, y becomes z, and so on), or we can permute within the slices. If you permute things around, the tensors are basically isomorphic; and you can apply the isomorphism on the rank decomposition, just like before. If we permute variables then we can permute them in the rank decomposition; same if we're permuting indices within the x's. So the rank decomposition with ε's of degree at most h will stay the same.

For (3), it's saying that if I take the Kronecker product of t and t' and I have a bound on the h-rank of t and h'-rank of t', then I can get a bound on the $(h+h')$-rank of $t \otimes t'$. So if we add tensors we take the max of the h's, and if we multiply we take the sum. This corresponds to the fact that if you add two polynomials you take the max of degrees, and if you multiply two polynomials you sum the degrees.

We're not going to prove (1) because it's the same proof as before. But we'll prove something — either (2) or (3) or both.

Proof of (2). Let's recall what a direct sum is — this means you have tensor t as a box in 3D, and then you put t' together with t without any overlap (they were in $K \times M \times N$ and $K' \times M' \times N'$, and now they're in $(K + K') \times (M + M') \times (N + N')$).

Let's say we have decompositions of t and t', and without loss of generality suppose $h \geq h'$ (so $h = \max(h, h')$). Let $r = R_h(t)$ and $r' = R_{h'}(t)$. Because $h \geq h'$ and we know $R_h(t) \geq R_{h+1}(t)$ from the earlier fact, we have $R_h(t') \leq r'$ as well. This is because I can expand

$$\varepsilon^{h'} t' + O(\varepsilon^{h'+1}) = \sum_{\ell=1}^{r'} a_\ell \otimes b_\ell \otimes c_\ell,$$

and I can multiply both sides by $\varepsilon^{h-h'}$ to get a new decomposition of $\varepsilon^h t + O(\varepsilon^{h+1})$ (with the same number of terms).

Now given this, we have that there's a h-rank decomposition for t with r terms, and an h-rank decomposition for t' with r' terms. So we ca write them out as

$$\sum_{\lambda=1}^{r} u_\lambda \otimes v_\lambda \otimes w_\lambda = \varepsilon^h t + O(\varepsilon^{h+1}) \quad \text{and} \quad \sum_{\ell=1}^{r'} a_\ell \otimes b_\ell \otimes c_\ell = \varepsilon^h t' + O(\varepsilon^{h+1}).$$

In $u_\lambda \otimes v_\lambda \otimes w_\lambda$, these have lengths K, M, and L; in $a_\ell \otimes b_\ell \otimes c_\ell$ they have lengths K', M', N'. And we want to put them in a space with dimensions $K + K'$, $M + M'$, and $N + N'$. So all we need to do is, given these vectors, construct new vectors with these new lengths such that when I sum these vectors, I get a rank decomposition for $t \oplus t'$.

This is pretty simple — for each $\lambda = 1, \ldots, r$, we define $\overline{u}_\lambda = \begin{bmatrix} u_\lambda & 0 \end{bmatrix}$ (where we have u_λ in the first K positions, and 0 in the next K' positions). We define $\overline{v}_\lambda$ and $\overline{w}_\lambda$ similarly. We also do the same with $\overline{a}_\ell$, where the first K positions are 0 and then we have a_ℓ; and we do the same for $\overline{b}_\ell$ and $\overline{c}_\ell$.

And now when I sum these two, we get

$$\sum_\lambda \overline{u}_\lambda \otimes \overline{v}_\lambda \otimes \overline{w}_\lambda + \sum \overline{a}_\ell \otimes \overline{b}_\ell \otimes \overline{c}_\ell,$$

they lie in the same space, and we get exactly

$$\varepsilon^h(t \oplus t') + O(\varepsilon^{h+1}).$$

So I get a rank-$(r + r')$ h-rank decomposition of the sum (I just sum the two decompositions, but I have to make sure they're in the same space, so I pad them with 0's appropriately). □

Because summing polynomials makes the degree the max of the two degrees, we get (2). When we want to prove (3), we're going to multiply the two rank decompositions (in a tensor product way).

Proof of (3). Just like before, we have decompositions $\sum_{\lambda=1}^r u_\lambda \otimes v_\lambda \otimes w_\lambda = \varepsilon^h t + O(\varepsilon^{h+1})$ and $\sum a_\ell \otimes b_\ell \otimes c_\ell = \varepsilon^{h'} t' + O(\varepsilon^{h'+1})$. And now we're going to look at when we multiply these. When we multiply the right-hand sides, we get

$$(\varepsilon^h t + O(\varepsilon^{h+1})) \otimes (\varepsilon^{h'} t' + O(\varepsilon^{h'+1})).$$

Using distributivity, this becomes

$$\varepsilon^{h+h'}(t \otimes t') + O(\varepsilon^{h+h'+1})$$

(for all the other terms involving at least one of the errors, I'll have degree at lst $h + h' + 1$).

So if I tensor-product the two decompositions on the left, I'll get an $(h + h')$-decomposition for the tensor product.

And what does this look like? If I want to tensor the things on the left-hand side, I can rewrite that as

$$\sum_{\lambda=1}^{r} \sum_{\ell=1}^{r'} (u_\lambda \otimes a_\ell) \otimes (v_\lambda \otimes b_\ell) \otimes (w_\lambda \otimes c_\ell).$$

Let's take a peek at what one of these looks like. With $u_\lambda \otimes a_\ell$, these were vectors of length K and K', and now this thing is a vector of length KK'. And its coordinates are defined by

$$(u_\lambda \otimes a_\ell)_{ij} u_{\lambda i} a_{\ell j}$$

(where $i \in [K]$ and $j \in [K']$).

And $u_{\lambda i}$ is some polynomial of degree at most h in ε, and $a_{\ell j}$ is some polynomial of degree at most h'. So when you multiply them out, you get a polynomial of degree at most $h + h'$ (when you multiply polynomials, you sum their degree).

So I get vectors of length KK', MM', and NN'; and the number of terms I get is rr'. So now I have rr' terms, which corresponds to the $R_h(t)R_{h'}(t')$ term; and the degree is $h + h'$. □

§20.3 Border rank vs. rank

So far, we've shown these properties; now we're going to use them. Before we use them, we'll give a simple lemma that relates border rank to rank — for any border rank decomposition, you can translate it into a rank decomposition, except that you blow up the number of terms. From then on, we'll show these approximate algorithms where you have ε can be converted into real ones.

Lemma 20.8 (Border rank to rank)
For every tensor t, if $R_h(t) \leq r$, then $R(t) \leq \binom{h+2}{2} \cdot r$.

So if you can find a border rank decomposition where the degree in ε is h, and it has r terms, then you can convert it to a true rank decomposition; except the number of terms is r times roughly h^2.

Proof. We'll write the same thing as we had before — say we have some h-rank decomposition

$$\sum_{\lambda} u_\lambda \otimes v_\lambda \otimes w_\lambda = \varepsilon^h \cdot t + O(\varepsilon^{h+1}).$$

This means if I look at the contribution from the left-hand side from terms where the degrees sum to exactly h, I get my tensor t.

So we can write $u_\lambda = \sum_{i=1}^{h} u_{\lambda i}\varepsilon^i$ and $v_\lambda = \sum_{j=0}^{j} v_{\lambda j}\varepsilon^j$ and $w_\lambda = \sum_{k=0}^{k} w_{\lambda k}\varepsilon^k$, and this thing says that

$$t = \sum_{\lambda=1}^{r} \sum_{i+j+k=h} u_{\lambda i} \otimes v_{\lambda j} \otimes w_{\lambda k}.$$

This happens to be the true rank decomposition (there are no ε's here).

And the number of terms is r times the number of triples $i + j + k = h$, which is exactly the $\binom{h+2}{2}$ term. (This is not too hard to prove; but the point is it's just a polynomial in h.) □

Remark 20.9. If your field is infinite, you can maybe make this into $h \log h$; but we don't really care, we just care that it's a polynomial in terms of h.

§20.4 Border rank to MM

Given this, now we're going to prove that you can use border rank to get MM applications — even though there's some ugly polynomials and so on, you can actually get true algorithms out of a border rank decomposition, with fewer terms than a true rank decomposition. And then we'll see some examples.

Last time, we showed that if we can get a bound on the rank of a matrix multiplication tensor — if for some constants m, n, and k we could show that

$$R(\langle m, n, k\rangle) \leq r,$$

then we would get

$$\omega \leq \frac{3 \log r}{\log(kmn)}.$$

So if we could bound the rank of some constant-sized rectangular MM tensor, we'd immediately get a bound on ω — just like with Strassen's recursive algorithm, you can get a recursive algorithm using the rank decomposition of the constant-sized tensor.

Today we'll see an analogous statement for border rank.

Lemma 20.10
If $\underline{R}(\langle m, n, k\rangle) \leq r$, then

$$\omega \leq \frac{3 \log r}{\log(kmn)}.$$

The border rank can be smaller than the rank, so this gives us more power — we can get better MM algorithms (as we'll see soon).

Proof. Define $Q = kmn$. Then we have that

$$\langle Q, Q, Q\rangle \cong \langle m, n, k\rangle \otimes \langle k, mn\rangle \otimes \langle n, k, m\rangle$$

(since we know when you take a Kronecker product of MM tensors, you just multiply their dimensions).

Now, here we have a bound on the border rank — let h be such that $R_h(m, n, k) \leq r$. Then because these are just permutations of the $\langle m, nk, k\rangle$ tensor, by Part (1) of the above lemma we have that

$$R_h(\langle k, m, n\rangle), R(\langle k, m, n\rangle), R_h(\langle nk, m\rangle) \leq r.$$

So $Q \times Q \times Q$ MM is a tensor product of 3 MM tensors whose h-rank is at most r. Now we can use Part (3) to get a bound on the border rank of the $Q \times Q \times Q$ MM tensor — that gives us

$$R_{3h}(\langle Q, Q, Q\rangle) \leq r^3$$

(in Part (3) we have $h + h'$, and here we have three terms all with the same h).

Now for every $s \geq 1$, we have

$$\langle Q^s, Q^s, Q^s\rangle = \underbrace{\langle Q, Q, Q\rangle \otimes \cdots \otimes \langle Q, Q, Q\rangle}_{s \text{ times}}.$$

(So we have some constant Q, but we can take it to any power we want — this is like Strassen's recursive algorithm.) Applying Part (3) again, we get

$$R_{3hs}(\langle Q^s, Q^s, Q^s\rangle) \leq r^{3s}.$$

(We had $3h$ before, and now we're doing s products.)

Now the 'border rank to rank' lemma on this gives a *rank* bound on $\langle Q^s, Q^s, Q^s\rangle$ of

$$R(\langle Q^s, Q^s, Q^s\rangle) \leq \binom{3hs+2}{2} \cdot r^{3s}.$$

The exact expression $\binom{3hs+2}{2}$ doesn't matter too much; h is a constant and s is our variable, so we'll just write this as $\mathsf{poly}(s)$.

So what we've shown is that for every s, we have

$$\omega \leq \frac{3 \cdot \log(\mathsf{poly}(s) \cdot r^{3s})}{\log Q^{3s}}.$$

Now we write this out and get

$$\frac{3 \log r}{\log Q} + \frac{3 \log \mathsf{poly}(s)}{3s \log Q}.$$

Now, Q is a constant, and $\log \mathsf{poly}(s) = O(\log s)$. So the second term is $O(\frac{\log s}{s})$.

So we have that for every s,

$$\omega \leq 3 \cdot \frac{\log r}{\log kmn} + O\left(\frac{\log s}{s}\right).$$

If we take $s \to \infty$, the last term goes to 0, and we get

$$\omega \leq \frac{3 \log r}{\log(kmn)}.$$

□

Recall that ω is a limit. In other words, if you want to get something close to ω, you take big enough s, and then you can multiply matrices in as close to this exponent as possible. Basically what's mattering is that once you have big enough matrices, the $\binom{3hs+2}{2}$ term doesn't matter (it's polynomial in the log of the matrix size, which means it's tiny), and you get basically the same bound as if you had a rank decomposition instead of a border rank one.

§20.5 An example — Bini et. al.

We'll see an example of how you can use this to get a better bound. The first paper that noticed you could use these weird other rank decompositions to get algorithms was by Bini et. al.; here's their example.

Consider the following problem: You have a 2×2 matrix, and you want to multiply it by another 2×2 matrix, and you want the output. So we have

$$\begin{bmatrix} x_1 & x_2 \\ x_3 & x_4 \end{bmatrix} \begin{bmatrix} y_1 & y_2 \\ y_3 & y_4 \end{bmatrix} = \begin{bmatrix} z_1 & z_2 \\ z_3 & z_4 \end{bmatrix}$$

This is what Strassen studied. But Bini et. al. said, what if we don't want z_4? So we just ask for z_1, z_2, and z_3. The corresponding tensor is $4 \times 4 \times 3$ (instead of $4 \times 4 \times 4$), and it's known to have rank 6.

What Bini et. al. showed is that actually its border rank is $5 < 6$, and you can use it to do matrix multiplication.

Here's a border rank decomposition:

- $P_1 = (x_{12} + \varepsilon x_{22})y_{21}$.
- $P_2 = x_{11}(y_{11} + \varepsilon y_{12})$.
- $P_3 = x_{12}(y_{12} + y_{21} + \varepsilon y_{22})$.
- $P_4 = (x_{11} + x_{12} + \varepsilon x_{21})y_{11}$.
- $P_5 = (x_{12} + \varepsilon x_{21})(y_{11} + \varepsilon y_{22})$.
- $\varepsilon P_1 + \varepsilon P_2 = \varepsilon z_{11} + O(\varepsilon^2)$.
- $P_2 - P_4 + P_5 = \varepsilon z_{12} + O(\varepsilon^2)$.
- $P_1 - P_3 + P_5 = \varepsilon z_{21} + O(\varepsilon^2)$.

(How they came up with this is a different story.)

We're missing the z_4 term, but it's interesting for the following reason. Imagine I wanted to multiply a 2×2 by a 2×3 matrix, so their output is 2×3. Then what you can do is you can use this algorithm *twice*, with a bit of overlap. So the X matrix stays the same, but for the Y matrix you use this algorithm twice. YOu use it once on the 2×2 block formed by the first two columns of Y, and a second time on the last two columns. Then your output from the first run (in yellow) is going to be one L-shape in the 2×3 output, and the output from the second run (in blue) is the complementary L-shape. So you've shown

$$R(\langle 2, 2, 3 \rangle) \leq 10.$$

Now we apply our lemma about ω from before; and what we get is that

$$\omega \leq \frac{3 \cdot \log 10}{\log(2 \cdot 2 \cdot 3)} = \frac{3 \log 10}{\log 12} \leq 2.78,$$

which is better than everything we've seen so far that just used a rank decomposition.

This was the first example. In their first paper, they said they got an approximation algorithm for MM with this exponent. But then Bini discovered this lemma (border rank to rank) and therefore that it gives us a *true* algorithm as well. So we get a true exponent of 2.78.

§20.6 The Schönhage τ theorem

This was later explored by lots of people — Strassen and Schönhage and so on — and led to the following theorem, which we'll write but not prove today. This led to a revolution in MM algorithms, which showed for the first time that $\omega < 2.5$ (because with 2.79 and so on, people started believing $\omega = 2.5$ — that's the next natural step between 2 and 3 — but they managed to get it below that).

This theorem has many names (another is the *asymptotic sum inequality*, and so on). The reason for the τ in the name is that there's some variable τ that appears in it.

This is a weird sort of thing. It says, suppose I give you a border rank expression for the direct sum of a bunch of MM tensors. Then from this, I can give you an algorithm and tell you exactly what the exponent ω should be.

Theorem 20.11 (Schonhage τ theorem)
Suppose that $\overline{R}(\bigoplus_{i=1}^{p}\langle k_i, m_i, n_i\rangle) \leq r$ (and $r > p$). Then $\omega \leq 3\tau$, where τ is the solution to the equation

$$\sum_{i=1}^{p}(k_i m_i n_i)^{\tau} = r.$$

So we're taking a bunch of independent copies of the MM problem, possibly with different dimensions ($r > p$ is a technical condition and not super important). These are some integers, and you multiply them and take their product, and take it to the τ and sum them up. There must exist some $\tau \in [0, 1]$ which is a solution to this; and if you take that τ, we have $\omega \leq 3\tau$.

Virginia will tell us one example of why this is useful, and next time we'll prove it.

Example 20.12
Consider $t = \langle k, 1, n\rangle \oplus \langle 1, m, 1\rangle$.

The first term is an outer product (of a length-k column vector and length-n row vector), and the second is an inner product (of a length-m row vector and length-m column vector). If you just want a rank decomposition, it's known that you can't do better than just computing the two separately — the rank of this is $kn + m$.

But if m is special — specifically $m = (k-1)(n-1)$ — then you actually get

$$\underline{R}(t) = kn + 1.$$

So if m has this form, then with only one extra term, you get to tack on a pretty large inner product — you basically don't pay for the inner product. This is very strange. Virginia has the decomposition in the notes; we can look at it.

What does that mean for our τ theorem? Let's take $k = 4$ and $n = 3$. In that case, $m = (k-1)(n-1) = 3 \cdot 2 = 6$. Then we get $\underline{R} = 4 \cdot 3 + 1 = 13$. So by the τ theorem we have $\omega \leq 3\tau$, where τ is the solution to

$$12^{\tau} + 6^{\tau} = 13.$$

You can use your favorite optimization software to figure out what this is, and you get $\omega \leq 2.57$. This is a huge drop from 2.78 to 2.57 — because somehow we found a border rank decomposition that very cheaply tacked on an inner product to an outer prdouct.

Now we'll say a few words about how you prove the τ theorem. Next time we'll take this huge sum for which we have a bound on the border rank, and tensor it up (taking a Kronecker product a bunch of times). Then

we'll zero out a bunch of terms, very carefully, so that in the end we have a single MM tensor that remains. We'll be able to bound how large it is. And we'll also have a bound on its rank, since we know when we take a Kronecker product of a bunch of tensors, we get a bound on their border rank. Then we can convert border rank into rank; and because we took a huge power, we'll be able to get rid of the $\mathsf{poly}(h)$ term (just like before) and get a rank decomposition, which we know corresponds to a recursive algorithm for matrix multiplication.

Student Question. *Do you personally believe $\omega = 2$?*

Answer. No. But who knows; she might change her mind. The reason she doesn't is we've reached a point where we've used certain techniques, and we've run out of imagination of what else to do, and we've proven limitations on those techniques showing we can't get to 2. So until we get new ideas, we'll be stuck with something bigger than 2. The limitations are on basically everything we know. But they basically are at 2.1; and Virginia would be happy with 2.1, because why not?

You can see there's lots of notation, but all we're doing is throwing polynomials at it; and then when we take large powers and look at big matrices, we get rid of the polynomials, so now we don't even have them anymore; they're just a way to get to better rank bounds. On Thursday we'll just do combinatorics; there won't even be polynomials (they'll be swept under the rug with the lemma we showed today).

§21 April 29, 2025 — Schönhage's theorem

We have two more lectures left. Today we'll talk about how to prove Schönhage's theorem. Before that, there's a tentative schedule for project presentations on Piazza; if you have issues and want to move your slot around, let Virginia know. The report is due May 6; if you need more time, also let her know. (She wants them by May 6 so that we have time to read them properly.)

The theorem we want to prove: suppose we have some constant-sized matrix multiplication tensors $\langle k_i, m_i, n_i\rangle$ (and a constant number p of them), and we have some bound on the

Theorem 21.1 (Schönhage's theorem)
Let $r > p$, and suppose that we know

$$\underline{R}\left(\bigoplus_{i=1}^{p}\langle k_i, m_i, n_i\rangle\right) \leq r.$$

Then we get $\omega \leq 3\tau$ where τ is the solution of

$$\sum_{i=1}^{p}(k_i m_i n_i)^{\tau} = r.$$

(We won't prove that such a solution τ exists, but it's true.)

In the first 20 minutes, we'll prove some tools needed to prove this.

§21.1 Restrictions

Suppose we have a tensor $t \in \mathbb{F}^{K\times M\times N}$, and a map $A : \mathbb{F}^K \to \mathbb{F}^{K'}$ (think of A as a $K' \times K$ matrix) and $B : \mathbb{F}^M \to \mathbb{F}^{M'}$ and $C : \mathbb{F}^N \to \mathbb{F}^{N'}$. We can take these three linear transformations and use them to

transform t into a new tensor $(A \otimes B \otimes C)t$ in $\mathbb{F}^{K' \times M' \times N'}$. We'll write this in two ways. One is as

$$t'_{abc} = \sum_{i,j,k} A_{ai} B_{bj} C_{ck} t_{ijk}.$$

So this is a linear combination of the t_{ijk}'s from the original tensor, but we're multiplying them by the corresponding entries of A, B, and C.

Another way to view this, if you don't view it in coordinate notation — suppose you have a rank decomposition of t as

$$t = \sum_{\ell=1}^{r} a_\ell \otimes b_\ell \otimes c_\ell.$$

For any rank decomposition, you can obtain a decomposition of t' as

$$t' = \sum_{\ell=1}^{r} (Aa_\ell) \otimes (Bb_\ell) \otimes (Cc_\ell).$$

Recall that in this notation, a_ℓ is a vector in $\mathbb{F}^K$; and then you multiply it by the matrix A, which transforms it to $\mathbb{F}^{K'}$ (and similarly with the other ones). Regardless of which rank decomposition you pick, this is well-defined — because when you look at the corresponding entries, you'll get the coordinatewise formula we wrote above.

But when we write it in this way, we immediately get the property that:

Fact 21.2 — If $R(t) \leq r$, then $R((A \otimes B \otimes C)t) \leq r$.

(This is because you can apply A, B, and C to each of the pieces of the rank decomposition.)

This is a generalization of what we talked about when we looked at permutations of tensors — you can think of A, B, and C as permuting the slices, if they're permutation matrices. But here we allow them to be arbitrary matrices.

Definition 21.3. We say t' is a restriction of t, written $t' \leq t$, if there exist A, B, and C such that

$$t' = (A \otimes B \otimes C)t.$$

So you take t and apply some linear transformations, and get a new tensor.

Fact 21.4 — If t' is a restriction of t, then $R(t') \leq R(t)$.

Student Question. *Why is it called a restriction?*

Answer. You take something, and you can make something of potentially smaller rank. There's also specialized restrictions — you have permutation restrictions, or ones where you just zero out certain parts. It's called a restriction because you can only lower the rank; you can't increase it.

When we talk about matrices, we like to talk about the identity matrix. There's a corresponding notion of an identity tensor, namely the tensor with 1's on the diagonal:

Definition 21.5. The identity tensor $\langle r \rangle \in \mathbb{F}^{r \times r \times r}$ is the one such that

$$\langle r \rangle_{ijk} = \begin{cases} 1 & \text{if } i = j = k \\ 0 & \text{otherwise.} \end{cases}$$

You can write it as

$$\langle r \rangle = \sum_{i=1}^{r} e_i \otimes e_i \otimes e_i,$$

where e_i is the vector with a 1 in the ith coordinate (and 0's everywhere else). This immediately means the rank of the identity tensor is at most r. But also, just like with matrices, its rank is *exactly* r.

Fact 21.6 — We have $R(\langle r \rangle) = r$.

Here's a little proposition we'll prove:

Proposition 21.7
We have $R(t) \leq r$ if and only if $t \leq \langle r \rangle$.

So the only way you can have rank at most r is if you are a linear transformation of the identity tensor.

Proof. The first thing is, clearly if t is a restriction of $\langle r \rangle$, then its rank is at most r (this is by what we wrote earlier — if $t' \leq t$, then $R(t') \leq R(t)$). But now we need to show the opposite. Suppose that $R(t) \leq r$; we want to show that $t \leq \langle r \rangle$.

If $R(t) \leq r$, then there exists some rank decomposition

$$t = \sum_{i=1}^{r} u_i \otimes v_i \otimes w_i$$

(so there's some summation of r rank-1 tensors). Now we'll define three linear transformations that, when you apply them on u, v, and w here, you're going to get $\langle r \rangle$.

So here's what we're going to do: Let

$$A = \begin{bmatrix} | & \cdots & | \\ u_1 & \cdots & u_r \\ | & \cdots & | \end{bmatrix}$$

(so the columns of A are the u_i's), and similarly B has v_i in the ith column, and C has w_i in the ith column.

Then e_i is the vector with a 1 in the ith coordinate, so when we multiply Ae_i, we'll have $Ae_i = u_i$. Similarly, $Be_i = v_i$ and $Ce_i = w_i$.

So then if we consider $(A \otimes B \otimes C)\langle r \rangle$, and we apply it to $\langle r \rangle = \sum e_i \otimes e_i \otimes e_i$, we get

$$\sum (Ae_i) \otimes (Be_i) \otimes (Ce_i) = \sum u_i \otimes v_i \otimes w_i. \qquad \square$$

So if you have rank at most r, then you're actually a restriction of the $r \times r \times r$ identity tensor; and if you are such a restriction, your rank is at most r. So this is an exact characterization of what it means for a tensor to have rank at most r.

§21.2 The ring of isomorphism classes of tensors

If you happen to read some algebraic geometry paper where they talk about tensors, they won't talk about coordinates and stuff. They'll talk about homomorphisms (which are just linear transformations), and they will completely forget about a tensor having this representation and they'll just say that if a tensor has rank r, it is just the $r \times r \times r$ identity tensor. More specifically, it's isomorphic to it:

Definition 21.8. We say t is isomorphic to t', written $t' \cong t$, if $t \leq t'$ and $t' \leq t$.

In this definition, the only way two tensors can be isomorphic is if they lie in the same space. So we'll define a new notion of isomorphism' that allows us to direct sum a bunch of 0's; this will allow you to embed t in any larger space you want.

Definition 21.9. We say $t \cong' t'$ if there exist all-0 tensors n and n' such that $t \oplus n \cong t' \oplus n'$.

So now they can lie in different spaces and still be isomorphic, because you can pad them with 0's.

Given this, we can say that 'the isomorphic classes of tensors form a ring.' This will allow us to do algebraic operations on tensors as if they were numbers, basically.

Fact 21.10 — The isomorphism' classes of tensors form a commutative ring.

All this means is the following:

- There is an additive identity — the all-0's tensor — such that $t \oplus \langle 0 \rangle \cong' t$.
- There is a multiplicatice identity — the rank-1 identity tensor — such that $t \otimes \langle 1 \rangle \cong' t$.
- We can move parentheses around with addition and multiplication — $t \oplus (t' \oplus t'') \cong' (t \oplus t') \oplus t''$, and $t \otimes (t' \otimes t'') \cong' (t \otimes t') \otimes t''$.
- Direct sums and Kronecker products are commutative, i..e, $t \oplus t' \cong' t' \oplus t$ and $t \otimes t' \cong' t' \otimes t$.
- There's distributivity — $t \otimes (t' \oplus t'') \cong' (t \otimes t') \oplus (t \otimes t'')$.

So now we can talk about tensors as objects like numbers. (The reason for *commutative* ring is the fact that the Kronecker product is commutative.) This is nice because we don't have to look at coordinates.

§21.3 More setup

As two pieces of notation:

Definition 21.11. For positive integers a, we write

$$a \odot t = \underbrace{t \oplus \cdots \oplus t}_{a \text{ times}} \quad \text{and} \quad t^{\otimes a} = \underbrace{t \otimes \cdots \otimes t}_{a \text{ times}}.$$

Also, two reminders from before:

Fact 21.12 — If $R(\langle K, M, N \rangle) \leq r$, then $\omega \leq \frac{3 \log r}{\log KMN}$.

Fact 21.13 — If $R_h(t) \leq r$, then $R_{ha}(t^{\otimes a}) \leq r^a$.

Recall R_h was an extension of rank where we used polynomials of degree up to h; then when we take ath powers, now we use polynomials with degree up to ha. We won't really recall exactly what this means; we'll just use these properties in the proof.

Now that we've done the setup, we're going to start proving stuff.

§21.4 A simpler version of Schönhage

We're going to start with a simple lemma that kind of looks like Schönhage but is simpler; it doesn't have border rank, and only talks about adding the same matrix multiplication tensor to itself (as opposed to ones of different sizes). Then we'll prove it, and use it in the proof of the full theorem.

Lemma 21.14

Let a, b, K, M, N be constants. If $R(a \odot \langle K, M, N\rangle) \leq b$, then for all $s \geq 1$, we have

$$R(a \odot \langle K^s, M^s, N^s\rangle) \leq \left\lceil \frac{b}{a} \right\rceil^s \cdot a.$$

So here we have ordinary rank instead of border rank, and we're taking a direct sum of a copies of the MM tensor.

So if you tell me you have a rank decomposition of a copies of $\langle K, M, N\rangle$, then I can give you a rank decomposition of a copies of much bigger tensors $\langle K^s \times M^s \times N^s\rangle$, and the rank grows basically as $(b/a)^s$. In particular, if instead of having a copies I looked at only one of the copies (this is a restriction of the a copies), I can get $K^s \times M^s \times N^s$ matrix multiplication with rank roughly $(b/a)^s$. So it's as if I'm getting an *average* in rank — if I can get a copies of a MM tensor with rank b, then I can essentially get a MM tensor with rank b/a (except a bigger one — because if I take $\langle K^s, M^s, N^s\rangle$, I would be getting b^s if I just used a single decomposition, but instead we can divide by a).

This is not hard to prove. We'll prove it by induction, using this isomorphism classes stuff.

Proof. We'll prove this by induction on s. The base case is $s = 1$; this is what is given to us, since

$$R(a \odot \langle K^1, M^1, N^1\rangle) \leq b \leq \left\lceil \frac{b}{a} \right\rceil \cdot a.$$

Now assume that we have

$$R(a \odot \langle K^s, M^s, N^s\rangle) \leq \left\lceil \frac{b}{a} \right\rceil^s \cdot a.$$

And we want to consider this, but with $s + 1$ on the left-hand side.

The first thing: We have

$$a \odot \langle K^{s+1}, M^{s+1}, N^{s+1}\rangle \cong (a \cdot \langle K^s, M^s, N^s\rangle) \otimes \langle K, M, N\rangle$$

(because the way Kronecker products of MM tensors works is that you multiply pointwise).

Now we apply the inductive hypothesis. The inductive hypothesis says that the rank of the LHS is at most the RHS. But that's equivalent to $a \odot \langle K^s, M^s, N^s\rangle$ being a restriction of the tensor $\langle \lceil b/a\rceil^s a\rangle$ (the identity tensor of dimension $\lceil b/a\rceil^s a$). So by that inductive hypothesis, we get that

$$a \odot \langle K^{s+1}, M^{s+1}, N^{s+1}\rangle \leq \langle \lceil b/a\rceil^s a\rangle \otimes \langle K, M, N\rangle.$$

Now we're going to move the a out to the second factor; so this is isomorphic to

$$\langle \lceil b/a\rceil^s\rangle \otimes (a \odot \langle K, M, N\rangle).$$

And then we can restrict $a \odot \langle K, M, N\rangle$ — we're given that its rank is at most b, which is equivalent to saying that it's a restriction of the $\langle b\rangle$ identity tensor. So we get that this is a restriction of

$$\langle \lceil b/a\rceil^s\rangle \otimes \langle b\rangle = \langle \lceil b/a\rceil^s \cdot b\rangle \leq \langle \lceil b/a\rceil^{s+1} a\rangle.$$

And so we've shown that

$$a \odot \langle K^{s+1}, M^{s+1}, N^{s+1}\rangle \leq \left\langle \lceil b/a\rceil^{s+1} a \right\rangle,$$

which is equivalent to the rank of the LHS being at most $\lceil b/a\rceil^{s+1} a$. □

Given this lemma, we're going to prove the full theorem. Before we do that, we'll give a corollary of this lemma — that assuming the same hypothesis, we can derive a bound on ω.

Corollary 21.15
If $R(a \odot \langle K, M, N\rangle) \leq b$, then $\omega \leq \frac{3\log\lceil b/a\rceil}{\log KMN}$.

So you get essentially the same bound as if you were proving the rank of $\langle K, M, N\rangle$ was b/a.

Proof. From the lemma, we get that for all s, we have

$$R(a \odot \langle K^s, M^s, N^s\rangle) \leq \left\lceil \frac{b}{a} \right\rceil^s \cdot a.$$

This also means that

$$R(\langle K^s, M^s, N^s\rangle) \leq \left\lceil \frac{b}{a} \right\rceil^s \cdot a$$

(if we can get a copies, then we can certainly get a single one). Then using our fact about bounds on ω from above, we get

$$\omega \leq \frac{3\log(\lceil b/a\rceil^s\, a)}{\log(KMN)^s}.$$

And we can write logs of products of things as sums, so this becomes

$$\frac{3s\log(\lceil b/a\rceil) + 3\log a}{s\log KMN} = \frac{3\log\lceil b/a\rceil}{\log KMN} + O(1/s).$$

(Both b/a and KMN are constants.) The $O(1/s)$ term goes to 0 as $s \to \infty$; and ω is a limit, so we get that

$$\omega \leq \frac{3\log\lceil b/a\rceil}{\log KMN}$$

(the second term goes away). □

This means if I get some bound on a bunch of copies of matrix multiplication, that bound gives me something which is equivalent to getting the ratio between the rank and number of copies, for the purposes of ω.

§21.5 Proof of Schönhage's τ theorem

Now we're going to start proving Schönhage's theorem. We'll need the two things given to us: One is that

$$\underline{R}\left(\bigoplus_{i=1}^{p}\langle k_i, m_i, n_i\rangle\right) \leq r,$$

and the other is the expression

$$\sum_{i=1}^{p}(k_i m_i n_i)^\tau = r$$

defining τ. We'll manipulate both of these.

The first thing says there exists some h such that the h-rank of this thing is at most r, i.e.,

$$R_h\left(\bigoplus_{i=1}^{p}\langle k_i, m_i, n_i\rangle\right) \leq r.$$

Now we're going to use the fact that $R_{ha}(t^{\otimes a}) \leq r^a$ — we're going to take a big tensor power of this thing and get a bound. So this means that for every s, we have

$$R_{hs}\left(\left(\bigoplus_{i=1}^{p}\langle k_i, m_i, n_i\rangle\right)^{\otimes s}\right) \leq r^s.$$

Now, last time we showed that you can convert h-rank into rank, so we're going to do that: we get that

$$R\left(\left(\bigoplus_{i=1}^{p}\langle k_i, m_i, n_i\rangle^s\right)^{\otimes s}\right) \leq r^s \cdot \mathsf{poly}(hs)$$

(when we convert border rank into rank, we take our rank bound, and multiply it by a polynomial in the degree — it was a square, but it doesn't matter what; and h is a constant, so this is really just a polynomial in s).

Now we have this rank expression; it's a bit bigger, but the dominant part is the r^s.

Now what we're going to do is take this huge power and expand it out as a huge direct sum, and we're going to argue about its rank and bound on ω using Corollary 21.15.

Let's take the thing

$$\left(\bigoplus_{i=1}^{p}\langle k_i, m_i, n_i\rangle\right)^{\otimes s}.$$

These things form a ring, so we can do operations in the following way. And we get that this is a huge direct sum

$$\bigoplus_{s_1,\ldots,s_p:\sum s_i=s} \frac{s!}{s_1!\cdots s_p!} \cdot \left\langle \prod_{i=1}^{p} k_i^{s_i}, \prod_{i=1}^{p} m_i^{s_i}, \prod_{i=1}^{p} n_i^{s_i}\right\rangle.$$

Here s_i is the number of times you see $\langle k_i, m_i, n_i\rangle$; the multinomial coefficient in the front is the number of ways to split s into chunks of $s_1, \ldots, s_p$. Because when you expand the LHS using the distributive property, you'll get a huge sm of properties.

This is what our tensor looks like. And what we want to do is identify one of these summands and ignore all of the others, and say that this summand already gives me something I can use Corollary 21.15 on. In order to identify which summand is very useful, I'm also going to expand our expression for τ in the same way, and discuss what that means.

Let's focus on the green expression for τ; that says

$$\sum_{i=1}^{p}(k_i m_i n_i)^{\tau} = r.$$

Similarly to what we did before, we'll take both sides to the sth power. And then we'll get the same sort of thing, except these are integers, so we have ordinary sums and products; so we get this huge thing

$$\sum_{s_1,\ldots,s_p:\sum s_i=s} \frac{s!}{s_1!\cdots s_p!} \cdot \prod_{i=1}^{p}(k_i m_i n_i)^{\tau s_i}.$$

Now we want to choose the $s_1, \ldots, s_p$ which maximize this summand — let $\overline{s_1}, \ldots, \overline{s_p}$ be the choices maximizing the entire summand

$$\frac{s!}{s_1!\cdots s_p!}\prod_{i}(k_i m_i n_i)^{\tau s_i}.$$

Now, how many summands are there? The number of summands is the number of choices of $s_1, \ldots, s_p$ that sum to s (that are nonnegative integers).

You can calculate this exactly — it's $\binom{s+p-1}{p-1}$ — but we don't need that; all we need is its order (particularly that it's a polynomial in s). Then we get that this entire thing is at most this maximizing value times the number of terms; so we get

$$r^s \leq \binom{s+p-1}{p-1} \cdot \left(\frac{s!}{\overline{s_1}! \cdots \overline{s_p}!}\right) \cdot (KMN)^\tau,$$

where we define

$$K = \prod_{i=1}^{p} k_i^{\overline{s_i}}, \quad M = \prod_{i=1}^{p} m_i^{\overline{s_i}}, \quad N = \prod_{i=1}^{p} n_i^{\overline{s_i}}.$$

We'll focus on these choices and the corresponding K, M, and N, and look at what this corresponds to for our big tensor. We said that

$$\left(\bigoplus_{i=1}^{p} \langle k_i, m_i, n_i \rangle\right)^{\otimes s}$$

had rank at most $r^s \cdot \mathsf{poly}(s)$. And if we look at the particular choices of the $\overline{s_i}$'s, we get that

$$R\left(\left(\frac{s!}{\overline{s_1}! \cdots \overline{s_p}!}\right) \odot \langle K, M, N \rangle\right) \leq r^s \cdot \mathsf{poly}(s)$$

— we look at our expansion on the right, forget about all terms except the ones defined by these choices of $\overline{s_i}$'s; and then we have $\langle K, M, N \rangle$ on the inside, and hte number of copies is this multinomial coefficient.

Now I have an expression that looks like Corollary 21.15, which takes in some number of copies of the same MM tensor and a rank bound for it. So now we're going to try to apply the corollary. For the corollary, we have

$$\frac{s!}{\overline{s_1}! \cdots \overline{s_p}!} = a \quad \text{and} \quad r^s \mathsf{poly}(s) = b,$$

so we're really interested in $\lceil b/a \rceil$. So let's look at that; we have

$$\frac{b}{a} = \frac{r^s \cdot \mathsf{poly}(s)}{\frac{s!}{\overline{s_1}! \cdots \overline{s_p}!}}.$$

Now, r^s is the business we had from before — we already had this expression

$$r^s \leq \binom{s+p-1}{p-1} \cdot \frac{s!}{\overline{s_1}! \cdots \overline{s_p}!} \cdot (KMN)^\tau.$$

So we just dump that in. The first term $\binom{s+p-1}{p-1}$ is just $\mathsf{poly}(s)$, so we'll dump it inside the $\mathsf{poly}(s)$; and we get

$$\frac{b}{a} \leq \frac{\mathsf{poly}(s) \cdot \frac{s!}{\overline{s_1}! \cdots \overline{s_p}!} \cdot (KMN)^\tau}{\frac{s!}{\overline{s_1}! \cdots \overline{s_p}!}}.$$

The big factorials cancel, and we get

$$\frac{b}{a} \leq \mathsf{poly}(s) \cdot (KMN)^\tau.$$

Now we use the corollary on this, and we get that

$$\omega \leq \frac{3 \log \lceil b/a \rceil}{\log(KMN)} \leq \frac{3 \log(\mathsf{poly}(s)(KMN)^\tau)}{\log(KMN)} = 3\tau + \frac{3 \log \mathsf{poly}(s)}{\log KMN}.$$

Now, the last term is

$$O\left(\frac{\log s}{\log KMN}\right).$$

So what we need to show is that $\frac{\log s}{\log KMN} \to 0$ as $s \to \infty$; and then we will exactly get 3τ, which is what we want.

So the only thing to show is to look at this thing $\frac{\log s}{\log KMN}$. So let's look at what KMN is. From our equation from before, we have

$$(KMN)^\tau \geq \frac{r^s}{\mathsf{poly}(s) \cdot \frac{s!}{\overline{s_1}! \cdots \overline{s_p}!}}.$$

And this multinomial coefficient is — you can think of it as the number of strings of length s over the alphabet $1, \ldots, p$, where there's exactly $\overline{s_i}$ letters i. In particular, this is at most the total number of s-length strings over this alphabet, which is p^s. So we get that

$$(KMN)^\tau \geq \frac{(r/p)^s}{\mathsf{poly}(s)}.$$

This means, if w etake logs, that

$$\log KMN \geq \frac{s \log(r/p) - O(\log s)}{\tau}.$$

And τ is a constant, so we get

$$\log KMN = \Omega(s).$$

(Here we're using the fact that $r > p$, so r/p is a constant bigger than 1, which means $\log(r/p)$ is a constant bigger than 0.) This means $\log KMN = \Omega(s)$. Now plugging this in, we get that

$$O\left(\frac{\log s}{\log KMN}\right) = O\left(\frac{\log s}{s}\right) \to 0 \quad \text{as } s \to \infty.$$

And so we get $\omega \leq 3\tau$.

Remark 21.16. So that's the τ theorem. Basically, what's happening is we're taking an sth power of this tensor, and inside that sth power, we're getting a *huge* MM tensor — so huge that $\log KMN$ is essentially s. So it's very big. And not only are we getting a huge single MM tensor, but we also get a very good bound on its rank — r^s times something negligible. So we get a huge MM tensor whose dimensions are essentially exponential in s, and we get a rank bound which is just a small exponential in s. And from that, we get a bound on ω.

There is some loss — you are disregarding a huge part of this tensor. But the loss is only polynomial — we're focusing on the largest summand (the largest volume), and the full tensor we're using has only a polynomial number of terms. So they're all bounded by the largest part, and it doesn't cost us very much to disregard them.

Remark 21.17. This theorem can be made more general (it doesn't have to be only about MM tensors), but it's much nicer when it's about MM — because tensor products of MM tensors are also MM tensors.

Remark 21.18. Basically what we did in the beginning was set up some tools so that we could forget about polynomials and stuff and just do combinatorics. And everything we do with MM — all the analysis — is just combinatorics. Once you extrapolate and generalize, border rank disappears because you can convert it to rank, and once you have rank it's just combinatorics.

§21.6 Forecast: Coppersmith–Winograd

Now Virginia will talk about what we're going to do next time.

We're going to talk about Coppersmith–Winograd. Their paper deals with two families of tensors. They're both parametrized by some integer q, which basically gives you their dimension. Virginia will draw those tensors for us, because they're simple.

The first is the small family of tensors. We'll draw them in matrix notation, though we can also write them out.

These small tensors have x-variables $x_0, \ldots, x_q$, and $y_0, \ldots, y_q$. The way they look is

$$\sum_{i=1}^{q} x_0 y_i z_i + x_i y_0 z_i + x_i y_i z_0.$$

So we draw a table with $x_0, \ldots, x_q$ on the columns (written at the bottom) and $y_0, \ldots, y_q$ on the rows (written bottom to top on the left); and we have 0 in the bottom-left, z_0 on the rest of the diagonals, and $z_1, \ldots, z_q$ in the bottom row and left column.

If you stare at these pieces long enough, recall that the coefficients in front of the z's are the things we're computing. If we focus on z_0, you're computing $\sum_{i=1}^{q} x_i y_i$. That's an inner product, also known as $\langle 1, q, 1\rangle$.

On the first term, for each i, you're computing $x_0 y_i$. This is a product of a 1×1 matrix x_0 by a vector $(y_1, \ldots, y_q)$; so this is $\langle 1, 1, q\rangle$. So what this looks like is

$$\langle 1, 1, q\rangle + \langle q, 1, 1\rangle + \langle 1, q, 1\rangle.$$

If these $+$'s were direct sums, we could apply Schönhage and get an amazing bound on ω. Unfortunately, they're not direct sums, because these parts share variables — for example, y_1 appears in the first part and the last part, and z_0 is shared in a lot of places, and so on. Because of this interference, we cannot just use Schonhage.

What we end up doing is taking this tensor, and taking a big power of it, just like we did with Schonhage. And we try, in that big power, to convert it to a direct sum of matrix products, by killing off some terms.

What do we know about this family of tensors? We call this tensor cw_q. We would love for its border rank to be the number of variables; this would be amazing. The number of variables is $q+1$. However, it's known

$$\underline{R}(cw_q) \leq q + 2.$$

(If it were $q + 1$ then we would get $\omega = 2$, but it is known that that is false for any q.)

It would be just as good for us if we could take cw_q and take some powers of it — if its border rank dropped to $(q+1)^{\bullet}$, this would be as good for us. Because if we took $cw_q^{\otimes n}$, we would have $(q+1)^n$ variables. And if we had border rank going to this, we could still prove $\omega = 2$.

But we don't know how to do that. Because we don't know what's going on with these tensors, C–W defined another family of tensors, called the big ones.

Remark 21.19. But the group-theoretic approach for MM is really about this small family — they try to recast this small family in terms of group algebras, and work with it and try to show $\omega = 2$ from there. They don't actually work with the big CW which we will define.

People believed somehow we'd be able to show the 'asymptotic rank' of these guys: the asymptotic rank of t is defined as

$$\lim_{n\to\infty} ((R(t^{\otimes n}))^{1/n}).$$

Really the reason we work with border rank is we don't know how to get a handle on asymptotic rank (if we could, we could do much better, because this gets smaller and smaller for many tensors).

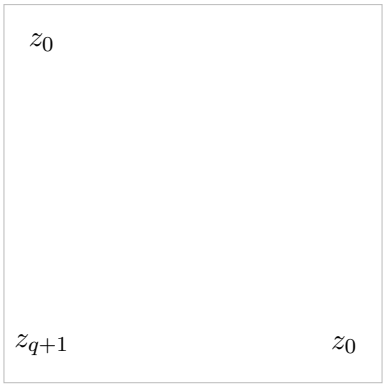
Now let's define the big CW family of tensors CW_q (again q is some integer). This is the same as the small cw, plus a few extra terms — we define

$$CW_q = cw_q + x_0 y_0 z_{q+1} + x_0 y_{q+1} z_0 + x_{q+1} y_0 z_0.$$

So this one has $q+2$ variables; in other words, it lies in $\mathbb{F}^{(q+2)\times(q+2)\times(q+2)}$. And its border rank is

$$\underline{R}(CW_{q+2}) = q+2,$$

exactly the number of variables; so it's optimal. But it's not as nice. The three extra things are $\langle 1,1,1\rangle$, basically scalar multiplications: and if we draw them out, it looks like



So we had the original MM from before, and we added on three scalar multiplications to it; and the rank is somehow still the same (it turns out you can fit these extras).

Roughly, here's why this is the case. There's a different version of this where you have the same thing, but instead of having z_0's on the NW-SE diagonal, you rotate them so they're on the SW-NE diagonal. This, we had this tensor polynomial multiplication before, that we gave a few lectures ago; and that one had basically the same values along the diagonals. This other version fo CW_q, some of these diagonals are hollowed out; there's a way to zero them out, which we'll talk about next time, so the border rank stays the same. And we know for polynomial multiplication, the rank si obtained via FFT and roots of unity, and it's exaclty the number of variables. So for polynomial multiplication, the number of variables is $q+2$, so rank is $q+2$, and when you hollow out the border rank si $q+2$.

And all the best MM algorithms we now of come from using one of these tensors. Next time we'll talk about how you do that. And we dont' know a better tensor (we've tried).

next time Virgnia will give us the techniques of getting something like this where you have interference (your tensors ovelrap in variables) to isolate a direct sum in a high power tensor without blowing up the rnak; and then you get a bound for ω using Schonhage τ.

§22 May 1, 2025 — Coppersmith–Winograd

Today Virginia will tell us about Coppersmith–Winograd.

Today our goal will be to prove $\omega < 2.404$.

§22.1 The small CW tensors

We'll do it using the small CW tensor. Virginia will remind us what that tensor is: cw_q is a tensor family where $q \geq 1$ is an integer. This is a tensor that lies in $\mathbb{C}^{(q+1)\times(q+1)\times(q+1)}$. So we have $q+1$ x-variables, $q+1$

y-variables, and $q+1$ z-variables. And the tensor is defined as

$$\sum_{i=1}^{q} x_0 y_i z_i + x_i y_0 z_i + x_i y_i z_0.$$

Each of these individual things is a matrix multiplication tensor (a very rectangular one). In the first, x is a single variable, so that's a 1×1 matrix, and you're multiplying it by a $1 \times q$ vctor; so this is $\langle 1, 1, q \rangle$. For the second term, y_0 is a 1×1 matrix and you're multiplying by a $q \times 1$ vector, so that's $\langle q, 1, 1 \rangle$. For the third, x and y are both length-q vectors and you're obtaining a single number, their inner product; so that's $\langle 1, q, 1 \rangle$.

§22.2 A special case of Schönhage

We're going to use a very special case of the Schönhage τ theorem, which looks much better than what we had last time:

Suppose you have a direct sum of p matrix multiplication tensors $\bigoplus_{i=1}^{p} \langle k_i, m_i, n_i \rangle$, and we have a bound on their border rank that

$$\underline{R}\left(\bigoplus_{i=1}^{p} \langle k_i, m_i, n_i \rangle \right) \leq r.$$

We define the 'volume' of a matrix multiplication tensor $\langle k_i, m_i, n_i \rangle$ as $k_i m_i n_i$.

A special case of Schönhage is when all your MM tensors have the same volume.

Corollary 22.1

Suppose that $\underline{R}(\bigoplus_{i=1}^{p} \langle k_i, m_i, n_i \rangle) \leq r$, such that $k_i m_i n_i = v$ for all i. Then

$$\omega \leq \frac{3 \log(\lceil r/p \rceil)}{\log v}.$$

You can derive this from the Schönhage theorem we got last time — we had $\omega \leq 3\tau$ where τ was the solution to $\sum (k_i m_i n_i)^{\tau} = r$. And if all $k_i m_i n_i$'s are the same, you gt $p v^{\tau} = r$, which gives you this bound on ω.

We'll just use this cleaer version, because with the small CW tensors, in the underlying tensors, all our summands have volume q.

§22.3 The main theorem and some consequences

We're going to try to prove the following theorem.

Theorem 22.2 (CW 1986)

If $R(cw_q^{\otimes n}) \leq c^{n+o(n)}$ for some c and q, then $\omega \leq \log_q(\frac{4c^3}{27})$.

And then, suppose we prove this. Then we can get the following corollaries.

If the theorem is true for $q = 2$ and $c = 2$:

Corollary 22.3

If $\underline{R}(cw_2) = 3$, then $\omega = 2$.

Why do we care about 3? For cw_2, the number of variables is $q+1 = 2+1 = 3$. The smallest that the border rank could be is the number of variables, which is 3. So if it so happens that the border rank of cw_2 is 3, then we immediately get $\omega = 2$.

Let's prove this using the theorem.

Proof. If this is true, this means there exists some h such that $R_h(cw_2) = 3$. This also then means that for every n, we have $R_{hn}(cw_2^{\otimes n}) \leq 3^n$ — remember that when we take tensor powers, you multiply the degree of the polynomial (which is h) by that power. Then we convert this into actual rank — we get

$$R(cw_2^{\otimes n}) \leq 3^n \cdot \mathsf{poly}(n)$$

(really it's a quadratic in hn, but h is a constant, so that goees away).

And if we have that, now we get to apply Theorem 22.2 with $c = 3$ and $q = 2$, and we get $\omega \leq \log_2(\frac{4\cdot 3^3}{27}) = \log_2 4 = 2$. □

So this is something Coppersmith and Winograd noted in their paper. Unfortunately, recently it was proven that $\underline{R}(cw_2)$ is not 3; it's 4. So you can't use this.

But you can apply the same sort of thing if you don't have the border rank, but still have that the rank of some nth power is 3^n. It might be that we still have $R(cw_2^{\otimes n}) = 3^{n+o(n)}$. Unortunately it's known that $R(cw_2^{\otimes 2}) = 16$, so the rank doesn't drop in the second power. But in some unpublished work, they were able to show that this rank (this is kind of called the asymptotic rank) — they can show that it's actulaly less than 4^n. You can show it's at most 3.9^n, actually. So the rank does drop below 4^n. The question is, does it drop to 3^n? If it does, we're in great shape and $\omega = 2$.

So it's possible one could prove $\omega = 2$ this way; we don't know.

Now let's give the second corollary.

Corollary 22.4

We have $\omega < 2.404$.

Proof. We know that $\underline{R}(cw_q) \leq q+2$. (We mentioned this last time.) So applying the theorem with $q = 8$, we know

$$R(cw_8^{\otimes n}) \leq (8+2)^n \cdot \mathsf{poly}(n)$$

(kind of like what we did above). This is 10^n, so you use the theorem, and get

$$\omega \leq \log_8 \left(\frac{4\cdot 10^3}{27}\right).$$

And then you plug this into your calculator, and get 2.404. □

So we just apply the known bound for cw_8 and get 2.404, which is much better than the previous known bounds on ω.

So what we want to do now is prove Theorem 22.2.

§22.4 Preliminaries

We're going to set up some preliminaries, and then we'll start analyzing this tensor.

§22.4.1 Zeroing out

The first thing is Virginia will remind us what a restriction is. The types of restrictions we'll use today are called *zeroing out*, or *combinatorial restrictions* — so we're not using arbitrary matrices this time.

Let's say you have some tensor $\sum t_{ijk}x_iy_jz_k$. What zeroing out means is literally you pick some x-variables, y-variables, and z-variables, and you set them to 0. This will kill off some of the terms of the tensor.

This 'combinatorially restricts' the terms of the tensor, and you only get some pieces.

Example 22.5

Suppose $t = x_0y_1z_1 + x_1y_0z_0 + x_0y_0z_0$. Let's say I set x_1 to 0. Then the middle term disappears, so I get $x_0y_1z_1 + x_0y_0z_0$. (So we killed off the middle term, and now we only have the first and last.)

Or you could set z_0 to 0, and then you'd only get the first term.

Or you could set them all to 0, and then you'd get the 0 tensor.

When you zero out variables, the rank and border rank can never increase (because you could just zero out the corresponding variables in the rank or border rank decomposition). And it can decrease the rank (e.g., if I set them all to 0, then the rank becomes to 0).

And not only does it preserve (i.e., not increase) the rank, but it's also a type of restriction — recall a restriction means you pick matrices A, B, and C and apply them to your slices. For zeroing out, these matrices are just diagonal matrices — A will be basically the identity matrix, but then you pick some of the diagonal entries and set them to 0. If I take the (i,i)th entry and set it to 0, this will kill off x_i. And similarly with B and C — they're basically identity matrices with some entries on the diagonal set to 0.

So this is a very special type of restriction. But it's also easier to think about because it's extremely combinatorial — we can select which variables to kill off and think about that combinatorially (we don't have to think about arbitrary matrices working on our x, y, and z vectors).

Here's a question.

Example 22.6

Look at the above tensor $t = x_0y_1z_1 + x_1y_0z_0 + x_0y_0z_0$. Is it possible to always restrict, just by zeroing out, to get any possible combination of the terms?

The answer is no — we can't only kill off the third term. If we kill off the third term, we're killing off either x_0, y_0, or z_0, and no matter which one we kill off, it'll kill off either the first term or the second as well.

So zeroing out only allows us to keep *some* subtensors, but not all. So it's a bit limited. But it's useful because they're restrictions, and they keep the rank and border rank bounded.

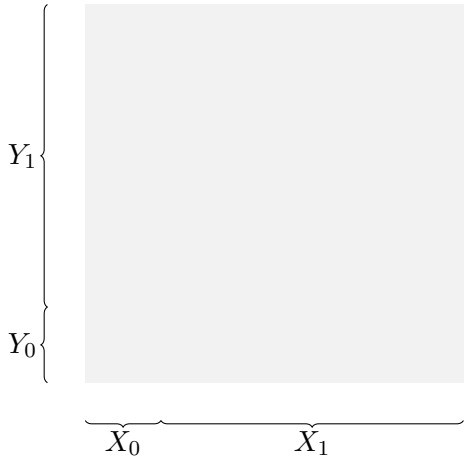
§22.5 Variable blocks

Now we're going to take our tensor cw_q, which is very nice and symmetric, and partition its variables in a particular way. Let's draw the tensor in matrix form: we have 0 in the lower-left corner, z_1, …, z_q in the bottom row and left column, z_0's on the SW-NE diagonal, and 0's everywhere else.

I'll create variable blocks: First, for the x-variables, I split into $X_0 = \{x_0\}$ and $X_1 = \{x_1, \ldots, x_q\}$.

Similarly, I'm going to split up the y-variables as $Y_0 = \{y_0\}$ and $Y_1 = \{y_1, \ldots, y_q\}$.

I also split the z-variables similarly as $Z_0 = \{z_0\}$ and $Z_1 = \{z_1, \ldots, z_q\}$.



I'll think of this picture as having an *outer structure* and *inner structure*. The outer structure is what the blocks look like:

$$\begin{bmatrix} Z_1 & Z_0 \\ 0 & Z_1 \end{bmatrix}$$

(with X_0 and X_1 written on the bottom, and Y_0 and Y_1 on the left from bottom to top).

There's nothing in (X_0, Y_0); (X_1, Y_0) contains Z_1; (X_0, Y_1) also cointains Z_1; and (X_1, Y_1) contains Z_0. So I'm just rewriting which variables appear in which parts.

The nice thing about how we've blocked the variables is that this outer structure looks kind of like the polynomial multiplication tensor, for which we have FFT and optimal rank. In particular, we were talking about the variables appearing on diagonals; this is what FFT exploints. And that's what the outer structure looks like.

The inner structure is what's inside each of the blocks. Each of those blocks corresponds to one of these $\langle 1, 1, q\rangle$, $\langle q, 1, 1\rangle$, or $\langle 1, q, 1\rangle$ — these correspond to $X_0Y_1Z_1$, $X_1Y_0Z_1$, and $X_1Y_1Z_0$ respectively. So the inner structure is matrix multiplication tensors of various sizes and volume q.

So outside is basically polynomial multiplication, and inside is matrix multiplication.

We're going to exploit this structure when we take cw_q and take it to a large tensor product, and we'll see what happens with these guys — the outer structure and the inner structure.

This might look arbitrary, but it's actually very natural. We'll now redefine some things about the tensor.

In general, if you want to get MM algorithms from some other tensor, you try to find some blocking of variables so you get an outer structure like this, with these diagonals, and then we can do things with it.

We'll define a subtensor of cw_q as follows.

Definition 22.7. For $a, b, c \in \{0, 1\}$, we define T_{abc} to be the subtensor of cw_q restricted to the variables in X_a, Y_b, and Z_c.

Example 22.8

- $T_{000} = 0$, because there are no z-variables in (X_0, Y_0).
- $T_{011} = \sum_{i=1}^{q} x_0 y_i z_i$, which is $\langle 1, 1, q\rangle$.

Claim 22.9 — We have $cw_q = \sum_{a+b+c=2} T_{abc}$.

So it's the sum of the T_{abc} tensors where $a, b, c \in \{0, 1\}$ sum to 2. This is very nice and important, and actually describes the diagonal structure of our outer structure.

§22.6 Powers of the CW tensor

The next thing we'll do is take cw_q to the nth power. For some intuition, we'll first take it to the 2nd power.

We've written it as

$$cw_q = \sum_{a+b+c} T_{abc} = T_{011} + T_{110} + T_{101}.$$

So now, if I take it to the second power, I get the 2nd power of this thing; so basically, I get

$$cw_q = \sum_{\substack{a+b+c=2 \\ a'+b'+c'=2}} T_{abc} \otimes T_{a'b'c'}.$$

Now, each of these guys T_{abc} are matrices of volume q; so each T_{abc} has volume q, and so does $T_{a'b'c'}$. When we multiply two MM tensors of volume q, we get a MM tensor of volume q^2.

Example 22.10

If we had $\langle q, 1, 1\rangle \otimes \langle 1, q, 1\rangle$, then you'd get $\langle q, q, 1\rangle$.

The second tensor power has variables whose indices are pairs of the variables of the original tensor — so $cw_q^{\otimes 2}$ has variables of the form $x_{p_1p_2}$, $y_{p_1p_2}$, $z_{p_1p_2}$, where these are pairs of indices of the original variables — i.e., $p_1, p_2 \in \{0, \ldots, q\}$. (We originally had variables indexed by $\{0, \ldots, q\}$; now we have variables indexed by pairs.)

And for example, if I take

$$\left(\sum x_0 y_i z_i\right) \otimes \left(\sum x_j y_0 z_j\right),$$

then I'll get a matrix product that looks like

$$\sum_{i,j} x_{0j} y_{i0} z_{ij}.$$

And this is your new MM, which is a $\langle q, 1, q\rangle$. So when I take a tensor to the second power, the variables get pairs of the original indices.

Now we're going to rewrite this — we'll call

$$T_{abc} \otimes T_{a'b'c'} = T_{aa',bb',cc'}.$$

That's because the original variables were split into blocks (X_0, X_1), (Y_0, Y_1), (Z_0, Z_1). In the second power $cw^{\otimes 2}$, you can also split them — you get this inherited blocking X_{ab} for $a, b \in \{0, 1\}$, where this means all the variables whose original indices — the indices are pairs, and the first index is in X_a and the second in X_b.

Now we'll define this in general for the nth power, and then we'll continue.

Let's look at $cw_q^{\otimes n}$. Then the variables will have indices which are sequences of length n — variables are indexed by sequences $i_1 \ldots i_n$, where i_j is an original index $i_j \in \{0, \ldots, q\}$. There are blocks of variables — the variables are blocked in X_I, Y_J, and Z_K, where I, J, and K are block index sequences — i.e., $I, J, K \in \{0, 1\}^n$ — such that a variable $x_{i_1 \ldots i_n}$ is in X_I if for every j, if $I_j = 0$ then $i_j = 0$, and if $I_j = 1$ then $i_j \in \{1, \ldots, q\}$. So we have long index sequences for both the variables (0 to q) and blocks (0 or 1), and it's just the inherited blocking.

And, kind of similarly to what we had before, we have

$$cw_q^{\otimes n} = \sum T_{IJK},$$

where now $I, J, K \in \{0,1\}^n$ are binary sequences of length n, and for every j, we have $I_j + J_j + K_j = 2$. Previously we had pairs of blocks, and now we have this. And T_{IJK} is the subtensor of $cw_q^{\otimes n}$ restricted to the variables in X_I, Y_J, and Z_K.

In the first power we only had single blocks, and now the blocks are indexed by sequences of length n. Whenever the jth value in a block sequence is 0, the variable better have a 0 in that value; and if it's 1, then the variable has a value from 1 to q.

Example 22.11

If I have the block $I = 0111$, then x_{0qq1} is in here, because it has a 0 whenever I has a 0 and something between 1 and q for the others.

Now what are these T_{IJK}'s? Before, each of those was a MM tensor of volume q. These guys are MM tensors of volume q^n (because we're multiplying n of them). So all these T_{IJK}'s are MM tensors of volume q^n.

And how many are there in the sum? The number of summands is 3^n — that's because for each of the n choices, there's only three T_{abc}'s you could pick (T_{011}, T_{110}, T_{101}). So there's 3^n summands.

If we wanted to apply Schönhage, we might think $p = 3^n$ and the volume is q^n, so... but the problem is we have a *sum*, not a *direct sum*. All these MM tensors are not disjoint — they share variables. So this is not a direct sum; instead, it's just a sum.

So what we want to do is select some variables to zero out, so that what remains are *independent* matrix multiplications. Then this sum will be a direct sum. We want to maximize the number of summands we get (hopefully close to 3^n). The volume will stay the same, and we want the rank (or border rank) to stay roughly the same. Right now, the border rank is $c^{n+o(n)} \leq (q+2)^{n+o(n)}$.

§22.7 A combinatorial problem

Now we're going to select some variables to zero out. We'll actually select whole *blocks* to zero out — we'll pick some blocks and kill off all the variables in the block. This will restrict to only some of the terms T_{IJK}. And this will define a purely combinatorial problem, which we'll say in a second.

We're going to define L to be a set of triples — the index triples of my variable blocks. In fact, we're going to work with $3n$th powers for simplicity — so in the above thing, we introduce a 3 everywhere we see n (i.e., we're considering $cw_q^{\otimes 3n}$ instead). Then we define

$$L_0 = \{(I,J,K) \mid I,J,K \in \{0,1\}^{3n},\ I_j + J_j + K_j = 2 \text{ for all } j\}.$$

Then my tensor is just the sum of T's over indices in this set L_0. What I want to do is *remove* some of these triples, so that what remains is a very big number of independent ones.

So let's define what independent means.

Definition 22.12. We say triples (I,J,K) and (I',J',K') are independent if $I \neq I'$, $J \neq J'$, and $K \neq K'$.

This means if I look at T_{IJK} and $T_{I'J'K'}$, they don't share any variables — because their variables are defined on completely different blocks of variables (X_I, Y_J, and Z_K vs. $X_{I'}$, $Y_{J'}$, $Z_{K'}$). So if I have them as a sum, they're actually a direct sum, which is great.

Goal 22.13. Select some sets $S_x, S_y, S_z \subseteq \{0,1\}^{3n}$ such that if I restrict L_0 to these sets, I have a large collection of independent triples.

So we're going to consider

$$L_0(S_x, S_y, S_z) = L_0 \cap \{(I, J, K) \mid I \in S_x, J \in S_y, K \in S_z\}.$$

And we want this to have independent triples (meaning no two things in here share variables). And we want the size of this thing to be maximized.

So we are selecting some subsets of the index sequences, so that when we do that, what remains is a set of triples that don't share variables.

What this corresponds to is zeroing out all the x-variables whose indices are not in S_x, all the y-variables whose indices are not in S_y, and so on. Because zeroing out doesn't increase the rank, we'll get the same rank. And if we can lower-bound the number of triples we have, then we're getting a huge direct sum of matrix products of volume q^{3n}.

So we can forget about tensors:

> **Question 22.14.** What's the maximum number of independent triples we can get by selecting subsets of the x, y, and z indices (as described above)?

We are losing somewhere, which was later fixed in later papers: When we look at this combinatorial problem and completely disregard tensors, we're only allowing ourselves to zero out *entire* variable blocks. So we're not looking at all possible zeroing out, only ones that take subsets of the blocks and zero them out completely. So there is some loss, but it makes the problem much cleaner.

§22.8 A first restriction

We're actually going to do something with L_0 — we'll get rid of any I, J, or K that doesn't have exactly n 0's and $2n$ 1's. These have length $3n$, and I'm going to kill off all the I, J, and K's that are not nicely structured.

So we're killing off all I, J, and K that don't have exactly n 0's and $2n$ 1's. Now we get

$$L = \{(I, J, K) \mid I, J, K \in \{0,1\}^{3n}, I_j + J_j + K_j = 2 \text{ for all } j, I, J, K \text{ have } n \text{ 0's}\}.$$

How many triples are in here?

The first thing to note is that whenever I and J are fixed, K is determined (by $I_j + J_j + K_j = 2$).

The answer is the multinomial coefficient

$$|L| = \binom{3n}{n, n, n}.$$

This is because you pick the 0's in I, and then from the 1's in I, you pick the 0's in J. So you pick I to have n 0's and then $2n$ 1's. And then whenever I has a 0, J must have a 1 (because you can't have two 0's). So all the 0s of J are in the $2n$ 1's of I; so we have $\binom{2n}{n}$ ways to pick J, and then K is determined. So you get

$$\binom{3n}{n} \cdot \binom{2n}{n} \cdot 1 = \binom{3n}{n, n, n}.$$

This is not far from 3^n — it's basically 3^n up to $o(n)$ in the exponent.

§22.9 3-APs

So we've restricted our combinatorial problem; and now we want to restrict it even more to get independent things.

Now we're goign to use a theorem about sets excluding 3-APs by Salem, Spencer, and Behrend. The reason is $I_j + J_j + K_j = 2$ is a linear equation, and we can use it to get lots of independent triples.

Theorem 22.15 (Salem–Spencer, Behrend)
For all large enough M, there exists a set of integers $S \subseteq [M]$ such that $|S| \geq M^{1-o(1)}$ and such that if $a, b, c \in S$ and $a + c = 2b \pmod M$, then $a = b = c$.

(The original theorem doesn't have a mod M, but it holds even without it.)

So there exists a large set of integers such that if any three integers in it form an arithmetic progression, then it must be trivial.

§22.10 The new goal

The combinatorial problem we want to solve now is:

Question 22.16. Select S_x, S_y, and S_z (subsets of $\{0,1\}^{3n}$) such that

$$L(S_x, S_y, S_z) = L \cap \{(I, J, K) \mid I \in S_x, J \in S_y, K \in S_z\}$$

has independent triples, and its size is maximized.

If we can get a bound on this, then we immediately get a bound on ω:

Claim 22.17 — If the number of independent triples you can get is at least $\varphi^{n-o(n)}$, then

$$\omega \leq \frac{\log \frac{(q+2)^3}{\varphi}}{\log q}.$$

Here we're using Schönhage. Our rank is $(q+2)^{3n}$, and we're dividing by the number of independent triples, which is φ^n. And then v is the volume of the MM tensors, and they all have volume q^{3n}. So you get

$$\omega \leq \frac{3 \log \frac{(q+2)^{3n}}{\varphi^{n-o(n)}}}{\log q^{3n}}.$$

The n's cancel (they become multiplicative factors of log), the $o(n)$ disappears in the limit, and you get the thing we want.

So we want to figure out what φ is; this is the entire idea. And we will use the Salem–Spencer theorem as a first step; and then we will refine our goal.

Here's our goal now. We take our triples, and we're going to have two steps to make them independent.

In the first step, we're going to define some hash functions on the indices — we'll hash I, J, and K into some range, and we'll use Salem–Spencer to kill off some of the variables using the hash function. So we'll use this to zero out a bunch of triples.

After this, we'll have blocked the triples into subsets, and what we'll achieve is that any two triples in *different* subsets will be independent. So we'll achieve a partitioning of the remaining triples into subsets $B_1, \ldots, B_k$ such that if two triples are in different subsets, then they're independent. This is the first step.

Then in the second step, we're going to use a greedy procedure to make each individual subset consist of independent triples.

After we do that, we'll have all the triples that remain will be independent; so we get a direct sum. And we'll be able to bound how many triples remain; that'll give us a value for φ, and then we get ω.

§22.11 Step 1 — Hashing

Now we'll do Step 1.

We're going to pick a value M, and it's going to be some prime in the interval $[3\binom{2n}{n}, 4\binom{2n}{n}]$. Why these values, we'll see in a bit.

Now we're going to define hash functions as follows. First we pick values $w_0, \dots, w_{3n}$, which are in $[M]$ and are chosen uniformly and independently at random. So you have this extra w_0; and then for each of the indices of I, J, and K, you have a w-value that you'll be multiplying it by.

Now, we define the following hash functions. We define h_x to be hashing the x-indices. So it'll take an index sequence I of length $3n$, and it'll has hit to

$$h_x(I) = \sum_{j=1}^{3n} w_j I_j \pmod{M}.$$

So you're taking a random linear combination of the values in the sequence I, weighted by the w_j's.

We define a y hash function similarly, as

$$h_y(J) = \sum_{j=1}^{3n} w_j J_j + w_0 \pmod{M}.$$

It's the same as the x-hash function (with a linear combination), but you add this extra w_0 term.

And then we define

$$h_z(K) = \frac{1}{2}\left(\sum_j (2 - K_j) w_j + w_0\right) \pmod{M}.$$

Here's a few things. First, for two different I and I', their hash values are pairwise independent:

> **Fact 22.18 —** If $I \neq I'$, then $h_x(I)$ and $h_x(I')$ are independent. Similarly, $h_y(J)$ and $h_y(J')$ are independent, and $h_z(K)$ and $h_z(K')$ are independent.

> **Fact 22.19 —** $h_x(I)$ and $h_y(J)$ are also independent.

This is because of the w_0 term in h_y. (And for I and I' it's because they differ in some index where one has a 0 and the other has a 1, and because of that w_j they're independent.)

Next, what does the condition $I_j + J_j + K_j = 2$ imply for us? If I look at $h_x(I) + h_y(J)$, what do I get?

> **Claim 22.20 —** We have $h_x(I) + h_y(J) = 2h_z(K)$.

Proof. We have

$$h_x(I) + h_y(J) = \sum w_j I_j + \sum w_j J_j + w_0.$$

And $I_j + J_j = 2 - K_j$, so we get

$$\sum w_j (2 - K_j) + w_0,$$

which is exactly twice of $h_z(K)$. □

So for any triples that we care about the hash values form an AP of length 3, by definition!

(The 1/2 is well-defined because M is a prime, so it's odd, so 2 has an inverse.)

So then what we're going to do is we're going to take our set excluding APs, for our favorite value of M, and we'll zero out all I, J, and K whose hash values are not in that set. So we zero out everything whose hash value is not in this set, and we define

$$L' = \{(I, J, K) \mid I,\ J,\ K \text{ have } n \text{ 0's}, I_j + J_j + K_j = 2, h_x(I), h_y(J), h_z(K) \in S\}$$

where S is the Salem–Spencer or Behrend set.

We won't be able to finish everything, but we'll say a few things. When we zero out everything that doesn't fall in the Salem–Spencer set, we know any triple that remains is hashed to the same value — if $(I, J, K) \in L'$, then we know $h_x(I) = h_y(J) = h_z(K)$. So they're all mapped to the same value — because this is a set that excludes 3-APs, and we know the hash values of any original triple forms a 3-AP, so anything that remains has to have all index sequences map to the same value.

So the remaining triples are *partitioned* based on which value they're mapped to; we define

$$B_s = \{(I, J, K) \in L' \mid h_x(I) = h_y(J) = h_z(K) = s\}$$

for each $s \in S$. And we know that any two triples that fall in different buckes cannot share variables, because their index sequences got hashed to different values. So we have achieved Step 1.

And the only thing we need to figure out is how many triples we've killed off, and how many remain — any two triples in different buckets are independent, and we just need to figure out how many we've killed off.

We'll do that by using the fact that our hash functions are pairwise independent. So we want to understand $|L'|$. Let's look at the expected number of triples that remain — so we're going to fix some $s \in S$ and try to figure out how large B_s is, i.e., $\mathbb{E}[|B_s|]$.

The first thing is, in order for a triple to be in this bucket, we need $h_x(I) = h_y(J) = s$ — we have

$$\mathbb{P}[(I, J, K) \in B_s] = \mathbb{P}[h_x(I) = h_y(J) = s].$$

Why don't we care about $h_z(K)$? Once we have I and J, K is determined; and by how we defined the hash values, we knwo that if $h_x(I) = h_y(J) = s$, then $h_z(K)$ is automatically also s.

And these two are pairwise independent, so this is

$$\mathbb{P}[h_x(I) = s]\mathbb{P}[h_y(J) = s] = \frac{1}{M^2}.$$

So the number of triples in expectation in B_s is actually just the total number of triples divided by M^2, which means

$$\mathbb{E}[|L'|] = \frac{|S|}{M^2} \cdot \binom{3n}{n, n, n}$$

(where the third thing is the total number of original triples). And $|S| = M^{1-o(1)}$, so this is

$$\frac{\binom{3n}{n,n,n}}{M^{1+o(1)}}.$$

So it's essentially a $1/M$-fraction of what remains. (It's in expectation, but once you have something in expectation, you can say there exist values attaining it.)

§22.12 Step 2 — greedily fixing buckets

So now we have to use a greedy procedure to make each of the buckets independent. The way we do that is actually silly. You pick each one of these buckets separately, and you try to greedily build a list of independent triples — you put a triple in the list. Then you take the next point and ask, can you add it to the list? If you can add it without making things dependent, then you do; otherwise you zero it out (together with a bunch of other triples). And you keep doing this.

So we fix B_s. And then we build a list — we start with Γ being an empty list. Then we try to add some triple (I, J, K) to Γ. If it's independent from everything in Γ, then we add it. Otherwise there's some triple that interferes with it; let's say that $(I, J', K') \in \Gamma$ (so they share I — it's symmetric, so they could share J or K as well).

So I can't add it. What do I do? I take J and kill off all triples that have J in the y-index — so I kill off all triples $(*, J, *)$. And I continue.

The thing is, you have to show you're not killing too many of these. The funny thing is this is very easy — what happens is, let's say the number I killed off here is R. Then Ihave actually killed off $\binom{R}{2} + 1 \geq R$ *pairs* of triples. And then you can show in expectation, the number of *pairs* of triples mapped to the same B_s is actually small. So once you've killed off this many triples, you've killed off at least this many pairs of triples; and that means the number of things you've killed off is small (because you can bound the number of pairs mapped to the same B_s).

And in the end, this step determines what you set M to (you'll see in the notes) — because the number of pairs of triples depends on M.

And then if you set M to be what we wrote earlier, the total number of things you get in the end is — if you set $M \approx \binom{2n}{n}$, then you get that the expected size of your list in the end — the number of independent triples — is at least $\binom{3n}{n} \cdot \frac{1}{M^{o(1)}}$. And once you have $\binom{3n}{n}$, then you bound this using Stirling's approximation to figure out what φ is, and you get

$$\varphi \sim \frac{27}{4}.$$

(The 27 is from this 3.) And once you get φ, then you apply this over in our bound and get what we wanted.