Algorithms Notes

Karl Pichotta

Spring 2013

Abstract

These are notes I've been taking for a graduate algorithms course taught by Greg Plaxton at UT Austin. They're taken primarily in-class, and probably only useful for my own personal reference.

Contents

1 Preamble: Useful identities

- For nonzero a, b, c, we have

$$a^{\log_b c} = c^{\log_b a} \tag{1}$$

- For large n, we have

$$\left(1 + \frac{1}{n}\right) \approx e \tag{2}$$

$$\left(1 - \frac{1}{n}\right) \approx e^{-1} \tag{3}$$

- For nonzero integer n, we have

$$n^2 = n + 2\binom{n}{2}. \tag{4}$$

  Both the algebraic and intuitive justifications for this identity should be pretty clear.

- For large n, we have

$$\sum_{i=1}^{n} \frac{1}{n} \approx \ln n + \gamma$$

  with γ the Euler-Mascheroni constant (about 0.58)

- For arbitrary positive n, k, we have

$$\binom{n}{k} \geq \left(\frac{n}{k}\right)^k$$

  can be seen from the equation:

$$\begin{aligned} \binom{n}{k} &= \frac{n!}{k!(n-k)!} \\ &= \frac{\prod_{i=0}^{k-1}(n-i)}{k!} \end{aligned}$$

  (FINISH ME)

2 Chernoff Bounds, the Tails of the Binomial

If we have a binomaial RV $X \sim B(n, p)$, then we have the following Chernoff bound, for $0 \leq \delta \leq 1$:

$$P(X \leq (1-\delta)np) \leq \exp\left\{\frac{-\delta^2 np}{2}\right\}$$

Supposing we throw $100n \log n$ balls, by focusing our analysis on a single bin and then applying Union Bound, we get that the probability that some $X_i \leq \log n$ is $\leq n^{-44}$, using the above bound.

Fact: Thinking about Hashing, suppose we throw n balls into n bins, and we want a bound on the max load of any given bin. We might hope it's $O(1)$, but it isn't. Instead, it is

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

with high probability.

Upper bound: Define $Z = \max_i X_i$, with X_i the number of balls in bin i. Consider bin 1. $E[X_1] = 1$, but this tells us nothing about the tail of the distribution, which is our interest here.

What then, is

$$Pr\left\{X_1 \geq c\frac{\log n}{\log\log n}\right\}$$

Framing it in terms of Chernoff bounds, we will want to use:

$$\delta = c\frac{\log n}{\log\log n} - 1$$

We need to use the Large Deviations bound (bound (2) on handout). This is the following: Suppose $X \sim Binom(n, p)$, then

$$Pr\left\{X \geq (1+\delta)np\right\} \leq \left(\frac{e^{\delta}}{(1+\delta)^{1+\delta}}\right)^{np}$$

For concreteness, we take $c = 100$. Our $np = 1$, so the RHS of the above chernoff bound is

$$... \leq \left(\frac{e^{\delta+1}}{\left(100\frac{\log n}{\log\log n}\right)^{100\log n/(\log\log n)}}\right) \leq \left(\frac{\log\log n}{\log n}\right)^{100\log n/(\log\log n)} \approx n^{-100}$$

WE inflate the numerator by adding an extra e.

Sidenote: Now, this funny log logn terms come from the fact that solving

$$x^x = n$$

gives you something very like $\log n/\log\log n$. This comes from canceling out lower order approximations to log.

Returning, by union bound, we argue that the probability that some bin gets $\geq 100\log n/\log\log n$ balls is $\leq n^{-99}$.

Lower bound: So that was an upper bound. The lower bound is trickier. Let $k = \varepsilon\log n/\log\log n$, with ε a small positive argument. We won't be able to just reason about bin 1 and then use union bound—there is a constant probability $(1 - 1/n)^n \approx e^{-1}$ that bin 1 gets 0 bins. Let E_i denote the event that bin i receives at least k balls. We want to show that

$$Pr(\cup E_i) \geq 1 - \frac{1}{n^c}$$

This is, reminder, a lower bound on the max. That is, we're showing that

$$Z = \Omega(\log n/\log\log n).$$

We're showing that $Pr(E_1)$ is small, but it'll be useful.

So consider $Pr(E_1)$. We have

$$Pr(E_1) \geq Pr(X_1 = k)$$

(since RHS entails LHS) This is equal to

$$\begin{aligned}
Pr(X_1 = k) &= \binom{n}{k} (1/n)^k \, (1 - 1/m)^{n-k} && (5)\\
&\geq \binom{n}{k} (1/n)^k (1/e) && (6)\\
&\geq \left(\frac{n}{k}\right)^k (1/n)^k (1/e) && (7)\\
&= \frac{1}{ek^k} && (8)\\
&\approx \frac{1}{n^{\varepsilon'}} && (9)
\end{aligned}$$

We have the third to last step by the following useful lower bound:

$$\binom{n}{k} \geq (n/k)^k$$

which you see by expanding the formula and reasoning about the various values of quantities above and below the line.

Now, by linearity of expectation, we expect $n^{1-\varepsilon'}$ bins to get at least k balls.

What we want, however, is a high probability bound. That is, we want a statement of the form "at least one of bins, whp, will get at least k bounds". We can't use a Chernoff bound, because the E_i's are not independent, so therefore the distribution isn't Binomial.

From a high-level perspective: we throw n balls into n bins, then ask if E_1 occurred. If it did, then we'll be happy, because all we want is one of the E_is to occur (NB $PR(E_1) \geq 1/(n^{\varepsilon'})$. If E_1 doesn't occur, then note

$$PR(E_2 | E_1^c) \geq Pr(E_2) \geq 1/n^{\varepsilon'}$$

and so on with all of the $Pr(E_i)$'s. We can therefore argue that the probbability that noE_i occurs is

$$\leq (1 - 1/n^{\varepsilon'})^n$$

We want to factor this last equation into

$$\begin{aligned}
... &= ((1 - 1/n^{\varepsilon'})^{n^{\varepsilon'}})^{n^{1-\varepsilon'}} && (10)\\
&\approx (1/e) n^{1-\varepsilon'} && (11)
\end{aligned}$$

Instead of having $1/e$ to some logarithmic quantity, we have it to some polynomial quantity. So this is much, much less than inverse polynomial bound, and we've more than satisfied our sharp threshold.

An exercise close to the one we did at the begnning—using Chernoff bounds, we can argue that the number of flips of a fair coin to get $\log n$ heads with high probability is

$$\Theta(\log n)$$

(we use Chernoff bound (1), as in the first example). This comes up in at least one way to analyze randomized Quicksort.

In last class, we saw that the expected number of comparisons is $\Theta(n \log n)$. We now argue that, with high probability, the number of comparisons is $O(n \log n)$. Note this is quite different: what we say now is that the probability that you exceed the expected runtime by a factor of, say, 50, is very small. Note these RVs aren't binomial, but we'll be able to bound their behavior with binomial RVs.

First, we'd LIKE to show the number of comparisons involving a specific key is $O(\log n)$ with high probability. This is not true—there is a $1/n$ chance of an element being the first pivot, and it gets compared to n elements.

So what we'll do instead is to use a charging scheme: we charge a comparison to a non-pivot. Whenever we make a comparison, "charge" the comparison to the nonpivot (each comparison involves a pivot and a nonpivot). It IS true that the charge to any key is $O(\log n)$ with high probability. Once we show that, it immediately follows that the total number of comparisons is $O(n \log n)$ WHP. HOw can we use the prvious result to convince ourselves of this?

3 Hashing

Let's think about Randomized quicksort. We know that randomized quicksort has expected $O(n \log n)$ behavior. (that is, there are no "bad inputs" for it in expectation). Note also that Randomized quicksort has $O(n \log n)$ runtime with high probability (that is, we'll give a nice bound on the probability of the runtime being asymptotically higher).

We can think about quicksort from the perspective of a particular fixed key, call it x. Recall the charging scheme: we "charge" each comparison to the nonpivot (which we can do, since each comparison is between a pivot and another value).

Claim: The key x gets $O(\log n)$ charge with high probability (the failure probability will be $1/n^c$ for arbitrarily large c. (If we prove this, then by Union Bound, we have the total charge to all n keys is $O(n \log n)$ WHP.)

Proof of Claim: We have a "herd" of keys, initially of size n, and select one to be the pivot. The pivot is remarkably lucky—it gets 0 charge. If x is chosen as the pivot, that's nice. If, on the other hand, x is not chosen as pivot, then it will get charge 1. Then the keys will be partitioned into two "herds", one of which will never again be compared with x.

Consider the process where we start with a positive integer n. We flip a coin. If we get heads, we set the number to an integer uniformly drawn from $[0, 3n/4]$ (floored). If we get tails, then we set the value uniformly to between 0 and $n-1$. So at each stage, the value gets smaller (by at least one). The process stops when the number gets to 0.

We use this process as follows. Picking a pivot at random, there is a 0.5 probability that the pivot sits in between $n/4$ and $3n/4$ in the sorted list (it sits in the middle half). There's also a .5 chance it's not in the middle. The number in the process is just the size of the "herd" x is in.

If we get a good pivot, then the herd value is at most $3n/4$ (imagine getting the rightmost pivot in the middle half, say). This is like flipping a head. In the worst case (flipping a tail), then the size of the herd decreases by 1.

The process can't involve more than some constant times $\log n$ heads: each head decreases the herd size by at least 3/4. (Prove this?) So x sits there and hopes for good pivot choices from its perspective. (This is basically the end of the proof.)

We don't have a binomial variable, but we can relate it to this simpler process defined above, and use a Chernoff bound on that.

3.1 Hashing with chained overflows

We have a hashtable, visualized as an array of buckets, numbered $0, \dots, k-1$. We have a has function

$$h(x) \in \{0, \dots, k-1\}.$$

In order to handle collisions, we can use a linked list to chain them together in the bucket. Searching for an item in the table later involves computing the hash and then traversing the linked list.

Suppose you put k things in. You're generally hoping that each bucket tends to have $O(1)$ elements. In the worst case, you traverse a linked list (and have $O(k)$ lookup). In the best case, we get $O(1)$ lookup.

What's the runtime of h? Supposing we have n keys, each of which is $10 \lg n$ bits. (Note we're using the RAM model: the word size of the machine we're programming is logarithmic in the input size in bits of the problem. So if we have a million-bit instance, we assume that we can manipulate words of log 1m in constant time.) So that means that our word size is $\Theta(\log n)$ bits. That is, these keys will fit into a constant number of words. We'll be relying on this later to assume various operations are $O(1)$.

So for now, we think of h as taking any key and mapping uniformly random from $0, k-1$. This is of course not quite right: h must be deterministic. However, we think of the hash for our purposes as choosing uniformly at random.

This relates directly to the bin/bucket problems we discussed previously.

Claim: The average time for a search is $O(1)$, assuming n buckets (with n keys).

No proof (prove this?). Basically, the vast majority of buckets will get very few elements.

On the other hand, what's the expected max search time? That is, has n keys into n buckets with our idealized hash. The worst-case search time is the longest linked-list we get. We looked at this last time: The max load is

$$\Theta\left(\frac{\log n}{\log \log n}\right) \; w.h.p.$$

So it's only marginally better than using a red-black tree.

3.2 Perfect Hashing

At a high-level, we use *Perfect Hashing* to obviate this non-constant load problem. That is the following:

We assume we're dealing with a *static* set of n keys. That is, we construct a hash-table structure that is specific for the particular n keys that we have at hand. As before, assume each key is $10 \log_2 n$ bits.

Desiderata:

- We want to construct a hash table with $O(1)$ worst case search time. NOTE this isn't just a matter of expectation: we want guarantees about the worst case.
- We also want to use $O(n)$ space (that is, our solution can't be "use $O(2^n)$ keys").
- Further, we want "fast" construction

A naive approach is the following: Repeatedly pick a new hash function until you find one inducing max load of $O(1)$. We can do this because we have static keys (we know them in advance). Eventually we'll find one with $O(1)$ worst case. However, this won't give us fast construction: we'll have to run this an exponential number of times in order to find this (I don't quite get the proof, but it involves one of the results we showed last time involving $\log n$).

We'll use a twist on this: we'll follow a similar approach, but our criterion for selecting a hash function will be looser. we'll use a two-tiered approach: What we'll do is count "collisions", and as long as we don't have more than n collisions, we'll call it good. So we have n buckets, but instead of a linked list in each bucket, we have a hash table at each of the n buckets. That is, we have one primary hash table, and n secondary hash tables.

We want the total size of all these tables to be $O(n)$. We do our primary hashing; based on how many elements hash into a location, each bucket has a hash table of that size. We'll have the size of a secondary hash table being quadratic in the number of elements there. So if we have 20 elements in a bucket, we'll have a hash table of size approximately 20^2. Why? We want to totally avoid collisions at the second level of hashing. The thing about quadratic size is that this is the threshold at which we'll suffer 0 collisions in the secondary hash tables (we can repick secondary hashes).

One concern is that since we're wasting space in the secondary tables, when we add the size of the secondary tables up, it'll be too large. So we need to show that the sum of the table sizes will be linear (intuitively, we'll show the vast majority of buckets get only a constant number of keys).

So OK getting to details. What is a "collision"? Mapping keys to buckets, we'll let Y_i denote the RV corresponding to the number of keys mapping to bucket i at the top level (for $1 \leq i \leq n$). Let X be the RV denoting the number of collisions. Then

$$X = \sum_{i=1}^{n} \binom{Y_i}{2}.$$

That is, if 10 keys are mapped to a bucket, that gives 10 choose 2 pairwise collisions. Note that

$$X = \Theta\left(\sum_{i=1}^{n} Y_i^2\right)$$

which is, magically, the total size of the secondary hash tables. So X is constant-factor-related to the space required for the secondary hash tables. (Note that it's good enough, then, to pick a hash function that induces at most $100n$ or $1000n$ collisions.)

Now, to get at $E[X]$, we express X as a sum of indicator variables. Define Z_{ij} to be the indicator variable taking 1 if the i, jth keys collide, and 0 otherwise. Note there are n choose 2 such vars. So

$$\begin{aligned} E[X] &= \sum_{i,j} E[Z_{ij}] && (12) \\ &= \sum_{i,j} 1/n && (13) \\ &= \binom{n}{2}/n && (14) \\ &= \frac{n-1}{2} && (15) \end{aligned}$$

where we appeal to the fact that we have an idealized hash function. This is less than 1/2 of our target of getting $\leq n$ collisions. (It is important that we choose a constant $> 1/2$ when defining our criterion for accepting a hash functions.)

Call a hash function "good" if it induces $\leq n$ collisions; "bad" otherwise. On average, the number of collisions is $n/2$. We can therefore bound the percentage of all hash functions that are "bad". If, for example, 90% of the has functions are bad, then the expected number of collisions would be at least $0.9n$. So we have that at most half of the hash functions are bad. Note that this is a huge overestimate: the only way this can happen (that is, we get the expected value we had before) is if all the bad hash functions have exactly n collisions and the good ones have 0.

So in our first phrase, picking a primary hash function, This is, in the worst case, like flipping a fair coin until you get a heads. So the expected number of trials is $O(1)$ (it is Geometric(0.5)). So, with high probability, the number of trials is $O(\log n)$ (proof?).

Once we have the top-level hash function, then, the Y_i values are determined. In bucket i, we use a hash table of size Y_i^2. We repeatedly pick hash functions for bucket i until we find one that gives no collisions at all.

That is, the secondary hash table at buekct i has Y_i^2 size, Y_i keys. The expected number of collisions is calculated using a similar approach before. We define $\binom{Y_i}{2}$ indicator variables, and the probability that one of those is 1 is $1/Y_i^2$. So defining C_i th enum of collisions at i, we have

$$E[C_i] = \binom{Y_i}{2}(1/Y_i^2) = \frac{Y_i - 1}{2Y_i} \leq \frac{1}{2}.$$

The analysis is identical for every bucket. So cool! We're done, this scheme works.

3.3 Realistic Hash Functions

The tricky thing here is that we've been assuming that we have these idealized hash functions that distribute uniformly, and we can generate them easily (and they're independent). What is it like in real life?

When we were talking about idealized hash functions, the associated family corresponds to the family of

$$n^{n^{10}}$$

hash functions (this is the number of functions from $10 \lg n$ strings to n things, I think?). Note if we represent these naively, then we use base-2 log num bits to specify an element. This is probematic, because this description of a function is of length $n^{10} \lg 10$ bits.

The key thing to note here (that'll get us around this) is that pairwise independence is sufficient for the family of hash functions we use. What do we mean? We mean the following: A family of hash functions $\mathcal{H}$ is pairwise independent if, for any distinct keys x, y, and any (possibly equal) bucket indices i, j, if the hash function h is drawn uniformly at random from $\mathcal{H}$, then

$$Pr(h(x) = i, h(y) = j) = \frac{1}{B}$$

with B the number of buckets.

This is sufficient because, in our analysis, we were only interested in pairwise indicator variables. We were never interested in any more complicated conditions.

We're interested in designing a family of functions mapping from $(10 \log_2 n)$-bit strings to $(\log_2 n)$-bit strings.

We saw that for analysis, we didn't need full independence, but just pairwise independence. This is because our analysis cared only about the number of pairwise collisions, which we can write as a sum of indicator variables X_{ij}, which is 1 iff i and j collide. Recall that $P\{X_{ij} = 1\} = 1/p$, with p the number of buckets.

In our analysis, in fact, it's fine to have approximate pairwise independence; that can give us OK bounds. So we want a function h such that, for two distinct keys $x \neq y$ (and any i, j, not necessarily distinct), then

$$Pr\{h(x) = i \wedge h(y) = j\} = O\left(\frac{1}{n^2}\right).$$

IN the above, i and j are bins; x and y are keys.

First, we'll pick a prime p a bit bigger than n^{10} (why?). The prime number theorem tells us that about, picking things around the neighborhood, we only have to try a logarithmic number of keys (in n^{10}) in expectation before finding a prime. (There is also a theorem indicating that there is definitely a prime between k and $2k$; in particular, this is n^{10} and $2n^{10}$).

So let's hash from $\mathbb{Z}_p$ to $\mathbb{Z}_n$. Consider

$$h(x) = [ax + b \mod p] \mod n$$

with a and b chosen from $\mathbb{Z}_p$. (Note that we have a and b because we want it to be the case that, under our analysis, there will be no "bad input"—if we have an adversary picking keys, they won't be able to pick bad keys if they know what the family looks like; if there were no a and b, then they would be able to do so. In other words, our family needs to have more than one hash function in it if we want to argue a small number of expected collisions.) This isn't quite uniform between 0 and $n - 1$, but we're quite close. The inner hash

$$ax + b \mod p$$

is, I think, actually uniform over $\mathbb{Z}_p$ (proof?). The outer hash, then, will be approximately uniform over $\mathbb{Z}_m$ (proof?). This satisfies the condition for approximate pairwise independence that we defined earlier.

So at the top level, we pick a p, then we repeatedly choose a and b; for each a and b, we check how the particular hash function works. We repeatedly do that until we get a good a and b; we store them, as they fully parametrize a hash function. We do this for each primary hash function and each secondary hash function.

4 Dynamic Programming

4.1 Examples

4.1.1 contiguous subarrays of booleans

Suppose we're given an $n \times n$ boolean array A. We want to find the side-length of a largest all-true contiguous square subarray of A. So if there's a 3×3 subarray of Trues, then the desired answer is at least 3.

Now, certainly this problem is solvable in polynomial time. For each side length k, there are only n^2 different places where the $k \times k$ subarrays top-left corner could be; we could just exhaustively check this for all k and compute the answer.

We can get a better polynomial bound using Dynamic Programming. What DP does is, instead of solving a single problem instance, find a bunch of problem instances to solve, and do so in such a way that the larger problem instances can use the computations performed for the smaller instances.

So let a_{ij} be the size of the largest all-true contiguous square subarray with lower right corner at location (i, j). This is a family of n^2 subproblems. Our answer to the original problem is just

$$\max_{i,j} a_{ij}.$$

Now, what is a good order to solve these n^2 subproblems? What we'd like is that, whenever we get to a particular problem in this ordering, we can very easily solve it in terms of the instances we've already solved. In other words, we want to be able to write a suitable reccurrence for the a_{ij}'s. Thinking about it, it's not too tough to see the following works:

$$a_{ij} = \begin{cases} 0 & \text{if } (i,j) \text{ entry is F} \\ 1 + \min(a_{i-1,j}, a_{i,j-1}, a_{i-1,j-1}) & \text{otherwise} \end{cases} \tag{16}$$

So filling in left-to-right row by row (or top-down column by column) should work. The important thing is that, when we get to a position, the spot above it, to the left of it, above and to the left of it, are filled in. (A small note is that we need to interpret out-of-bounds indices as 0 above.)

So the above algorithm runs in $O(n^2)$ time, which is a nice, much-faster polynomial-time algorithm than the brute-force poly-time algorithm.

4.1.2 Rod-cutting problem

Commonly, Dynamic programming yields polynomial-time algorithms for problems where the brute-force technique is exponential. For example, suppose we have a rod of length n inches ($n \in \mathbb{Z}^+$). We can cut it into any integer-length pieces (such that they sum to n). For every i, we have a price p_i that we can sell a rod of length i for. The problem, then, is to cut up the rod into pieces in such a way that we maximize the total prices for the pieces.

When we cut the number up, we get a multiset of positive integers; in the language of number theory, this is a *partition* of n. If the number of partitions of n were sufficiently low, we could just generate them and calculate their value; however, the number of partitions grows superpolynomially. Consider the numbers

$$1, 2, \dots, \lceil 2\sqrt{n} \rceil - 1, \lceil 2\sqrt{n} \rceil.$$

Pair up the first two numbers and the last two numbers:

$$1, 2, \dots, \lceil 2\sqrt{n} \rceil - 1, \lceil 2\sqrt{n} \rceil.$$

For the first $\lceil 2\sqrt{n} \rceil + 1$ elements, you can either partition them into the two outermost pieces or the next innermost pieces.

We repeat for $3, 4$ and the next two inner numbers on the right. For each of these, we eat up $\sqrt{n}$ of the rod. We'll have $\sqrt{n}/2$ such decisions for what to do with the pieces of length $\sqrt{n}$, and we'll have about

$$2^{\Omega(\sqrt{n})}$$

ways to partition the rods up. (In fact, it ends up being $2^{O(\sqrt{n})}$ too.) So the number of partitions of n is pretty clearly superexponential.

We can solve this, however, with a simple one-dimensional dynamic program. For each i, let v_i be the maximum value of a rod of length i. We will use the following recurrence for v_i in terms of the smaller v_j's. For a partition of i, there will be some length ℓ of the last partition. So

$$\begin{aligned} v_0 &= 0 \\ v_i &= \max_{1 \le \ell \le i} v_{i-\ell} + p_\ell \end{aligned}$$

We get $O(n^2)$ time of this: we have n subproblems, but the ith subproblem takes $O(i)$ time to solve (summing over these subproblems uses the familiar $\sum_i i^2 = O(n^2)$ identity).

4.1.3 Longest Common Subsequence

We have two sequences X, Y of symbols over some finite alphabet. Writing X as

$$X = x_1 x_2 \dots x_n,$$

we define a subsequence of X as a subset of entries x_i, concatenated in order. We want to find the largest integer k such that there are subsequences in X and Y of length k such that both subsequences are the same.

The brute-force approach here is exponential: it's pretty easy to see (there are 2^n subsets of n sets).

We use the familiar two-dimensional DP solution. Let X_i denote the length-i prefix of X:

$$X_i = x_1 \dots x_i,$$

and similarly with Y_j. Now, let a_{ij} be the length of the LCS of X_i and Y_j.

Supposing $|X| = m$ and $|Y| = n$, we have $m \times n$ subproblems. The final answer to the question is a_{mn}. So we have the base cases

$$\begin{aligned} a_{0j} &= 0 \\ a_{i,0} &= 0 \end{aligned}$$

for all i, j. For the recursive case, we have

$$a_{ij} = \begin{cases} a_{i-1,j-1} + 1 & \text{if} x_i = y_j \\ \max\{a_{i,j-1}, a_{i-1,j}\} & \text{if} x_i \neq y_j. \end{cases} \tag{17}$$

4.1.4 The Partition Problem (a pseudopolynomial algorithm)

This problem is NP-hard. Suppose we have n positive integers $x_1, \dots, x_n$. We want to know whether the x_i's can be partitioned into two multisets of equal sum.

We give a DP algorithm. First, let $\sum x_i = 2s$. (Note if the sum is odd, we return that there is no such partition). To determine s, it' a natural thing to consider computing

$$a_{i,j} = \begin{cases} T & \text{if there's a subseq. of } x_1, \dots, x_i \text{ summing to exactly } j \\ F & o.w. \end{cases} \tag{18}$$

Note $\forall i$, $a_{i,0} = T$. Further, $\forall j > 0$ $a_{0,j} = F$. When $i, j > 0$, we have

$$a_{i,j} = \begin{cases} a_{i-1,j} & \text{if } j < x_i \\ a_{i-1,j} \vee a_{i-1,j-x_i} & o.w. \end{cases} \tag{19}$$

NB we only really need the first term for the technicality of the subscript on the RHS disjunct being negative.

Now, the number of table entries is $O(nS)$. Is this polynomial time? Intuitively, no—S can be very large. We argue that the algorithm is not polynomial; to do so, we need to find one family of inputs where the runtime is not upper-bounded by a polynomial function of the input in bits.

What do we mean by polynomial time? Well, we upper-bound the runtime of an algorithm according to a polynomial function of its input in bits. What's the input of this like? Well, consider an input where each x_i is an n-bit integer. The input size for this problem is $\Theta(n^2)$ bits (we have n n-bit numbers). For us to claim that we have a polynomial runtime, we need that, for such inputs, the runtime is polynomial in n. S, however, is $O(n2^n)$, since an n-bit integer can be as large as 2^n in value (and we have n of them). So, more importantly, S can be $\Omega(n2^n)$. So this algorithm, which kinda looks like it's polynomial, actually isn't polynomial in the input size.

However, sometimes these sorts of algorithms are useful. If we know, for example, that each of the numbers is at most n^3, then we have that S is at most n^4, and we have a polytime algorithm. We call algorithms like this *pseudopolynomial*: they are polynomial in the input size if the input integers are represented in unary.

5 Greedy Algorithms

5.1 A Scheduling Problem

Suppose we have n tasks, each with a positive integer deadline and an execution requirement. So an input may look like

task	Deadline	execution requirement
A	8	5
B	6	2
C	9	2

We want to know if we can meet all the deadlines in a nonpreemptive schedule? So, in the above, suppose we run B at time 0 and C at time 2; both of these take time 2, and have met their deadlines. If we run A next, though, it'll miss its deadline. However, if we run in the order BAC, we get a feasible schedule.

We may want to try to get at this by Dynamic Programming, but we don't need it: a greedy solution ends up sufficing. We use *earliest deadline*, breaking ties arbitrarily. This will yield a feasible schedule if one exists.

We want to argue that the answer to the greedy algorithm is right iff there's a feasible schedule. Clearly, Left to Right is easy. The argument is R to L, then:

If there's a feasible schedule, then we need to show the greedy algorithm finds it.

So take a feasible schedule. Assume the feasible schedule has no "spaces": there's no advantage to idle time, so everything is "squeezed" as far as possible to the left. Suppose that we have a schedule $ABCD$ of four elements, in that order. Suppose that C has an earlier deadline than B; earliest deadline would have chosen the order $ACBD$, rather than $ABCD$.

We can modify the schedule by leaving everything the same except for B and C, swapping the latter two. Note the new schedule is clearly the same length. We want to show that $ACBD$ is also feasible. Clearly C meets it deadline still—it's moved earlier. We need to show that B is meeting its deadline. Since, by assumption, C had a more stringent deadline than B, it must be the case that if B finishes when C did before, it certainly meets its deadline, since it meets C's deadline. In other words, before, with $ABCD$, C was meeting its deadline; we know that B's deadline is later than C's, so in $ACBD$, B definitely meets its deadline.

We apply the same logic to the new problem instance to argue that it's feasible (eventually we will get to the earliest-first schedule).

5.2 A variation of the scheduling problem

Suppose that, in addition to the deadline and execution requirements, each task has a positive profit $p_i > 0$ associated with it. Our goal is to find a max-profit feasible subset (that is, we won't in general use the entire set of tasks). What we'll do is combine the greedy approach with dynamic programming.

We first order tasks by deadline (in nondecreasing order). We then consider all prefixes with respect to that particular ordering. The DP's subproblem is $a_{k,t}$, defined as the max profit we can obtain with a schedule using only the first k tasks and time $\leq t$. We can write a recurrence fairly straightforwardly (left as an exercise).

The runtime of the above will be polynomial if the execution requirements are polynomially-bounded. That is, if every task has a task that is $O(n^c)$ for some c. Without such an assumption, we could get an exponentially-sized table in our DP

5.3 Matroids

Captures a large class of greedy algorithms. Consider the MInimal Spanning Tree problem, in particular, Kruskal's greedy algorithm. One way to convince ourselves that Kruskal's algorithm is correct is to reason directly about minimal spanning trees. Another is to frame the algorithm as a matroid problem, and use a general property about Matroids.

A Matroid is a pair

$$M = (S, \mathcal{I})$$

with S analogous to the vertices in a graph: it's just a set. $\mathcal{I}$ is called the *independent sets* of S: $\mathcal{I} \subseteq 2^S$ (elements are subsets of S). We need two more properties:

1. **Hereditary Property**: if $A \in \mathcal{I}$ and $B \subseteq A$, then $B \in \mathcal{I}$.
2. **Exchange Property**: If $A, B \in \mathcal{I}$ and $|A| > |B|$, then there is some $x \in A \setminus B$ such that $B \cup \{x\} \in \mathcal{I}$.

The hereditary property tells us that removing elements from an independent set yields an independent set. In particular, this tells us that $\emptyset \in \mathcal{I}$.

Define a maximal independent set as an independent set such that, if you add any additional elements to it, it will no longer be independent. The exchange property tells us that all maximal independent sets will have the same cardinality. A maximal independent set is sometimes called a **basis**

In a **weighted matroid**, each element $x \in S$ has an associated weight $w(x)$. In applications, we often want to compute a maximum (or minimum) weight maximal independent set. For example, eventually we'll have a minimal spanning tree representing a minimum weight maximal independent set (the maximal independent sets will correspond to the spanning trees).

5.4 Matroid Greedy Algorithm

Suppose we want to do maximization.

- First, sort the elements of S in nonincreasing order of weight (if we are minimizing, we'll sort in the other direction).
- Initialize $A := \emptyset$
- For each $x \in S$, in the order determined in the first step:
  - If $A \cup \{x\} \in \mathcal{I}$, then $A := A \cup \{x\}$.

And that's it. Thinking about Kruskal's algorithm, the set $\mathcal{I}$ will be the set of acyclic graphs—the conditional above corresponds to the conditional in that algorithm.

How do we show this algorithm is correct? First, index the elements of S $1, \ldots, n$ according to the ordering in the first step. Call the weight for element i in this ordering w_i. For each i, calculate some max-weight maximial independent set B. Mark down if element i is in B or not. Also, for each i, consider A, the set computed at that step by the algorithm. What we need to do is prove, for each step, A is a maximal weight independent step. Similarly, look at whether elem i is put in A.

Look at the first part where the sets A and B differ. It can't be the case that the elem i is in B but not in A (proof? It has somethign to do with the hereditary property, not sure what). So the first place where the two differ is that an element i must be in A and not in B. WE note the first place i where A and B differ. Construct $A' \subset A$ from all the before i in the ordering, and also elem i. Now, we know $|B| \geq A'$, since $|A'| \leq |A|$, and $|B| = |A|$, by the exchange property.

We repeatedly apply the exchange property to add elements from B to A'. The exchange property tells us that we can add an element from B to A' and get an independent set, so long as $|A'| < |B|$. This process terminates exactly when $|A'| = |B|$.

When we're done growing A', B gave A' all its elements from $i+1$ onwards except for one of them. In other words,

$$A' = (B + i) = j$$

for some j. That is A' has elem i while B doesn't, and A' deosn't get some $j > i$. However, since the elements are ordered by weight, and $w_j \leq w_i$, we can conclude that

$$w(A') \geq w(B).$$

Now we play the whole game over, but with B replaced by A'. This gets us "one step" in the right direction, and we iterate this process until we end up with $A' = A$, at which point we can argue that A was a max-weight maximal independent set.

5.5 Some examples of Matroids

- **Uniform matroid**: Let $|S| = n$. Fix $0 \leq k \leq n$. $\mathcal{I}$ is the set of all subsets of S of size $\leq k$. This is a matroid. The hereditary property is pretty obvious. The exchange property is also pretty clear: if we have A and B such that $|A| < |B|$, then there's clearly an element in B that's not in A such that adding it to A will result in an element of cardinality $\leq k$.

- **Partition matroid**: Fix a partition of S into $\{S_1, \ldots, S_k\}$. Define $\mathcal{I}$ to be the set of all subsets $A \subseteq S$ such that $\forall i |S_i \cap A| \leq 1$.

  A basis (maximal independent set) for this matroid would be a set that chooses exactly one element from each S_i. The number of such bases is

  $$\prod_{i=1}^{k} |S_i|.$$

  The hereditary property is pretty clear. The exchange property is a bit trickier, but also pretty clear.

- **Scheduling problem**: we have n tasks numbered from 1 to n. Task i has a positive integer deadline d_i, a positive weight w_i (the profit of the task, say), and an execution requirement of 1 (unit tasks). We're scheduling these tasks on a shared resource.

  What is the maximum-weight schedulable subset of the tasks (in general, we might not be able to meet all tasks of the schedule)? This is a bit complex: Suppose there are 2 tasks with deadline 1, and two tasks with deadline 2. We can schedule at most one of the former, but two of the latter (but only if we schedule neither of the first two).

This problem ends up having a useful matroid structure, allowing us to use the matroid greedy algorithm.

Define the set S as the set of n tasks. $\mathcal{I}$ will be the schedulable subsets of the tasks. If this works, our optimal solution will certainly be a maximal independent set (this isn't necessarily true if we have negative weights I don't think).

Now, to show $(S, \mathcal{I})$ is a matroid. The hereditary property is pretty clear. The exchange property is a bit trickier. Suppose $A, B \in \mathcal{I}$, with $|A| > |B|$. We proved earlier that A is schedule iff any ED schedule (???) of A is feasible (wtf does this mean?). Define a "canonical" ED schedule as one of the ED schedules: break ties in a particular manner. That is, we're constructing a schedule of a task A; we list tasks in increasing order of their index or something like that, guaranteeing a unique tie-breaking schedule. The schedule, then, will be some ordering

$$A = x_{i_1}, x_{i_2}, \ldots, x_{i_{|A|}}$$

and

$$B = x_{i_1}, x_{i_2}, \ldots x_{i_{|B|}}.$$

Note that there will be no "empty spaces" in the schedule.

Let's focus on the last task appearing in A that isn't a part of B. Define z to be this tasks: the last task in the canonical ED schedule of A that does not belong to B. To show the exchange property, we need to show there's something in A that can be added to B. We'll claim that z is such an element; that is, $B + z \in \mathcal{I}$ (that is, $B + z$ is schedulable).

- **Case 1**: z is the last task in the ED schedule of A. Because B is shorter, we can just tack z onto the end of B; it will surely meet its deadline, because it will start at least as early as it started in A. This isn't necessarily an ED schedule, but is feasible.
- **Case 2**: The ED schedule of A ends with $z, y_1, \ldots, y_\ell$ with $\ell \geq 1$ (that is, z isn't the last element of A's ED schedule). What we do is jimmy around with the schedule of B to arrive at a feasible schedule for $B + z$. Leave the schedule before y_1 alone; put z in the spot where y_1 is. Find y_2 in B's schedule, and put y_1 in it; similarly, put $y_{\ell-1}$ into the schedule where y_ℓ was, and tack y_ℓ onto the end (recall, these are the occurrences of $y_1, \ldots, y_\ell$ in B's ED schedule. Now, y_ℓ in the new schedule for B is no further to the right than it is in A's schedule. Similarly, the furthest to the right that $y_{\ell-1}$ could possibly be is where it is in A's schedule. So this is a feasible schedule (KBP: i don't see why $y_1, \ldots, y_\ell$ are in B's schedule. Maybe they don't have to be?).

- **Scheduling with release times**: a more complicated version of the above. Assume each task also has a positive integer **release time**. Previously, we assumed all tasks were available at time 0; we restrict that assumption now: tasks become available to executable at a certain point in time, and can only be scheduled past that time.

  Even with this additional constraint, we can construct a matroid structure (we don't here).

- **Transversal matroids**: even more general. Say we have a bipartite graph (U, V, E) (with U, V the two sets of vertices such that $\forall(u, v) \in E$, u and b aren't both in U or V). Consider the matroid $(U, \mathcal{I})$, with every $S \in \mathcal{I}$ a "matchable" subset of U, defined below. A **matching** is a subsets of the edges E that induces degree at most 1 on every vertex. That is, in a matching has no vertex incident on more than 1 edge (that is, every vertex has degree ≤ 1). Given a matching, call a vertex **matched** if it's incident on one of the edges in the matching. A subset U of vertices is **matchable** if there exists some matching matching every $u \in U$.

  So our matroid is $(U, \mathcal{I})$, with U the same set in our bipartite graph $(U, V, \mathcal{I})$ (proof that this is a matroid?). Relating to the last scheduling problem: Consider U the set of tasks, and V the set of times during which a task can start. So supposing task i has deadline of 5 and a release time of 2 (and takes 2 time units), we'll add edges from $i \in U$ to $2, 3, 4$ in V. Given this setup, every matchable subset corresponds to a valid scheduling.

  The hereditary property is easy: if a set of tasks is matchable, of course any subset is matchable.

  The exchange property is tricky: say we have $A, B \in \mathcal{I}$, $|A| > |B|$. Let M_A be a matching matching exactly A (which exists by assumption), and let M_B be a matching for B. That is, the set of nodes matching in M_A is exactly A (so $|M_A| = |A|$), ditto B. We need to show that there is some M matching $B + x$, for some $x \in A \setminus B$. Think of these matchings as sets of edges.

  Consider the graph $G \equiv (U, V, M_A \Delta M_B)$ (taht is, the symmetric difference of M_A and M_B: edges in exactly one of M_A, M_B). Because M_A is such that each node is of degree at most 1, we have that $M_A \Delta M_B$ induces degree of at most 2. G consists of vertex-disjoint paths and cycles (the cycles are of even length; each cycle would alternate between an edge in M_A and and edge in M_B—you can't have two consecutive edges from M_A, because this would give us a node of degree 2).

  We cannot have only cycles: since we know that $|M_A| > |M_B|$, we must have some paths (since every cycle draws an equal number of edges from M_A and M_B, I think). In particular, there must be at least one odd-length path that begins and ends with an edge in M_A. Note that the paths, like the edges, alternate from edges in M_A and edges in M_B. So assume WLOG that this path starts in U and ends in V (it's odd-length, and a bipartite graph, so an odd-length path can't begin and end in U or V). Call the node in U that the path starts at x.

We can obtain a matching for $B+x$ by making a slight hack on M_B. Take the odd-length path above; get rid of the original edges from M_B, and add the edges from M_A that were touching the same nodes. This doesn't lose matchability (every one of the nodes on the path is still connected to exactly one edge, this time the edge from M_A), and also now x is matched.

It's worth noting that the computational problem to solve the matroid greedy algorithm is trickier for this formulation: we have to determine matchability for arbitrary subsets, which is a little tricky.

- **Graphic matroids**: Fix an undirected graph $G = (V, E)$. Define $M = (S, \mathcal{I})$, with $S = E$ and $\mathcal{I}$ the set of all acylic subsets of E. The hereditary property is pretty obvious: removing edges won't add cycles.

  We don't have time for the exchange property in class.

6 Minimal Spanning Trees and Matroids

Recall that, given an undirected graph $G = (V, E)$, we construct a graphic matroid from it by the set $(E, \mathcal{I})$, with $A \in \mathcal{I}$ iff A is acyclic. Here, the Maximally independent set are exactly the spanning forests; if G is connected, then it is a spanning tree.

Note that a **minimal dependent set** of a matroid $(S, \mathcal{I})$ is a dependent set (that is, a subset of S not in $\mathcal{I}$) such that if we remove any element from it, we get an independent set. If we have a graphic matroid, then a minimal dependent set will be a cycle (we can't throw any edge out of the set without getting something that's acyclic).

6.1 A theorem and some corollaries about Matroids

Lemma: let $M = (S, \mathcal{I})$ be a weighted matroid. Let A, B be distinct max-weight bases of M. Then there exists a sequence $A_1, \ldots, A_k$ of max-weight bases of M such that $A = A_1, B = A_k$, and for every i with $1 \leq i < k$, we have

$$|A_{i+1} \setminus A_i| = 1.$$

That is, A_{i+1} and A_i differ by a "swap":

$$A_{i+1} = A_i + x - y$$

with x and y having equal weight (since they're all max-weight).

Proof: How do we prove this? One way is to use something similar to our proof of correctness for the matroid greedy algorithm. We order the elements (x_i, w_i) by decreasing order of weight, and then we line up A and B, marking whether $x_i \in A$ and $x_i \in B$. We look at the first element such that $x_i \in A$ but $x_i \notin B$, and, we obtain B' by growing an A^* somehow (how?!), starting with

A^*, and growing to a basis via repeated application of the exchange property with B. So we'll end up with B', which will differ from B by a single swap. There'll be one element that B does have but that B' doesn't have; other than that, B' (which is, I think, what you get after constructing as many A^*s as you can) and B will be identical.

Since A and B are both max-weight bases, what can we conclude about the basis B'? We have that $w(B') \geq w(B)$; we can get B' from B by exchanging the elements that differ, and the one in B' is at least as heavy as that in B (why?!). We cannot have $w(B') > w(B)$, since B is a max-weight basis.

So we will have obtained a third max-weight basis that agrees on a longer prefix with A than B did. We can do this again, getting a max-weight basis B'' differing from B' by a single edge swap; eventually, we will end up with something equal to A. We will therefore have exhibited a sequence of the desired form: Something starting at B and ending up at A, where each sequence is a max-weight basis differing from the previous by a single element.

Corollary to above: all max-weight bases have the same distribution of weights (that is, taking just the element weights and sorting it will yield the same list). (You might think that, e.g., you can have two minimal spanning trees such that one has an edge of weight 10 and 5, and the other has instead two edges of weight 8 and 7; this corollary rules out that possibility.)

Another corollary: Every max-weight basis is a possible output of the matroid greedy algorithm (that is, will be produced if ties in weight are resolved appropriately). You might think that, in Kruskal's algorithm, for example, that you can have a graph with a couple MSTs such that one of them will never be produced; this isn't possible, by this corollary. There will be a tie-breaking that will cause every MST to be produced. It's pretty easy to see what tie-breaking is appropriate: break ties by giving priority to elements in our desired max-weight basis.

A third corollary: if the weights are all distinct, then there is a unique max-weight basis. Cool!

6.2 Matric Matroids

There is another class of matroids called **"matric matroids"**, defined in terms of a given matrix A. Define such a matroid $M = (S, \mathcal{I})$, with S the columns of A, and $\mathcal{I}$ the sets of linearly independent columns of A. You can't model all matroids as matric matroids, but a bunch of the classes of matroids discussed above can be modeled as matric matroids.

It's useful, in general, to consider these special classes of matroids, rather than considering matroids in full generality, because we can often get faster algorithms for specific types of matroids.

7 Amortized Analysis

Amortized analysis is typically used in settings where the cost of operations is nonuniform. We ask, if we perform any k operations, what sort of bounds we can put on performance. So we may have some operations that are much more expensive than others, but nevertheless we may be able to bound the overall cost as being less than the pessimistic estimate (where we upper-bound every operation by the worst-case performance).

There are different methods of amortized analysis:

1. **Aggregate method**: Simply sum up the cost of all the individual operations to bound the total cost of a sequence of operations. This is the most basic approach, but can get kind of messy.

2. **Accounting method**: Suppose we're trying to prove a theorem that any sequence of k operations will take $O(k)$ time. Then, when an operation is performed, we'll give an operation a constant amount of dollars in order to do it. For example, we'll give an operation \$10 to fund its execution, if we're trying to bound the operations at amortized \$10. The actual semantics are that an operation uses the money we give it; if it uses less than we give it, it puts the rest of the money into different piles of money in the data structure. When an expensive operation takes a lot, then it uses the leftover piles of money from the preceding operations.

   If the money we inject into a system is enough to pay for all the operations, then we can get bounds on the running time.

   Note that it'll frequently make sense to convert this approach to the potential approach, described below (where the potential corresponds to "amount of money left in data structure").

3. **Potential method**: (sometimes, potential function method): A more general method, emphasized more in this course. We define a potential function mapping the state of the data structure to a (usually nonnegative) number. The amortized cost of an operation will be defined as the actual cost plus the change in potential.

   So if an operation changes the state of the data structure from D to D', then the change in potential is $\Phi(D') - \Phi(D)$, with Φ the potential. We'll use the potential function to make sure the amortized costs all look similar, making it easy to add up the amortized cost.

   This seems weird—what we really care about is the total cost, rather than the total amortized cost (which is this funny function of the actual cost that we made up). However, summing things up makes it make sense:

   Suppose the state of our data structure before/after each of our operations is $D_0, \ldots, D_k$. When we sum the total amortized cost, we get the potential telescoping: the sum of the $\Delta\Phi$'s is

$$\phi(D_1) - \phi(D_0) + \phi(D_2) - \phi(D_1) + \ldots = \phi(D_k) - \phi(D_0)$$

which gives us that the total amortized cost is the total cost plus the final potential $\Phi(D_k)$ minus the initial potential $\Phi(D_0)$.

Usually, the initial potential is 0, and the final potential is typically nonnegative; this means that the total amortized cost, according to this method, will actually be an overestimate of the total cost (since total amortized cost = total cost + final potential). So if we can place an upper bound on the total amortized cost, that will also give us an upper bound on the total cost.

7.1 Three Examples

7.1.1 Push/multipop stack

We have a cheap operation, push (push a single item from the stack), and a potentially expensive one, multipop, which pops k times (taking time proportional to k). Multipop(k) can only be performed on a stack with at least k elements.

So a push has actual cost 1, and multipop(k) will have cost k. What is the cost of a sequence of n operations starting with the empty stack?

- **Naive approach**: At all times, we cannot have more then n elements on the stack. No single operation costs more than n: at most, we can pop n elems at any time. Since we have n operations, the total cost is $\leq n^2$.

  This is a very weak upper bound: we'll see that our handful of expensive operations must be offset by a bunch of cheap operations.

- **Aggregate Method**: The total cost of all multipops must be $\leq$ the total cost of all pushes. If we ever multipop off 10 items, we can only do that if, previously, we'd pushed all of them on, at a total cost of n. Since the total cost of all pushes is $\leq n$ (we perform at most n pushes of cost 1), we have that the total cost is at most $2n$.

- **Accounting method**: Give the system \$2 for each push, and \$0 for each multipop. We'll want to show that the pushes have amortized cost 2 and the multipops have 0 amortized cost; this would imply that the total amortized cost is at most $2n$.

  When a push happens, it has to take one dollar; it puts the extra dollar onto the pile. At any point, a stack of m elements will have \$$m$ sitting around. In fact, we can think of each element keeping its dollar, and so we can easily prove that each element has one dollar associated with it, and so we'll be able to fund every operation.

- **Potential method**: Define ϕ of a stack to be the number of elements on the stack (notice that this is basically the same as the accounting argument). Then it's easy to show, using the argument we just gave, that we can bound the cost of operations linearly.

7.1.2 Incrementing a binary counter

We have a counter that has the number 0 initially, and we have only one operation: increment counter. We increment in the most elementary way: start at the LSB, flip and possibly carry, and continue until you hit a 0. So the increment operations are occasionally expensive (if you have to flip a lot of 1's to 0's), but they're usually cheap. We'd hope that, over a large sequence of k increments, our total cost would be $O(k)$.

Concretely, our counter has an infinite number of bits going off to the left. The first increment will take cost 1; the next will take 2 (we have to flip 2 bits); the third will flip 1 bit; the next 3; and so on.

- **Aggregate method**: the total cost of n operations will give us taht roughly 1/2 the operations will have cost 1 (every other number is even); 1/4 of the ops will cost 2; 1/8 will cost 3, and so on. So the total cost is

  $$(n/2) + 2(n/4) + 3(n/8) + \ldots \leq \sum_{i \geq 0} \frac{i}{2^i} = 2$$

  (the last equivalence is somewhat nontrivial)

- **Accounting method**: \$2 / operation. We'll need to maintain a different invariant from the stack: we might think that we'd have a dollar per bit, but that won't end up working. We instad have \$1 for each 1-bit in the counter: if our number is 001011, then we have \$3. It's not tough to see this works: incrementing the counter, if we have k bit flips, we had to convert $k - 1$ 1's to 0's; we grab that dollar going around, and then deposit our extra dollar on the 1.

- **Potential method**: ϕ will is the number of 1s present in the counter. The amortized cost is the actual cost $+\Delta\phi$. So note that the counter always looks like 01^k for some k (with some stuff before it); an increment flips that to 10^k. So the change in potential is $1 - k$: The potential before is however many 1's there are; the potential after is $1 - k$ minus that (we flip k bits and change 1 to a 0). So the actual cost is $k + 1$; the change in potential is $1 - k$; thus the actual cost + $\Delta\phi$ is 2. We can thus bound the cost of n increments by $2n$.

  Note this is basically also an accounting argument.

7.1.3 Dynamic Array

The canonical example: suppose we're using an STL vector or a Java ArrayList. The underlying implementation doesn't know how much space is needed; we'll reserve space, but from time to time, the application will exceed the bounds, and a large cost will be incurred. So sometimes a single push can be very expensive, but we'll want to determine that the total cost of a sequence of operations is, say, $O(k)$.

We want space usage to be proportional to the number of elements in the array; that is, we don't want to just allocate way more space than we'll ever need and call it a day. Further, we'll want to ensure that we're using at least a constant fraction of our currently allocated array. Further, we'll want contiguous elements in memory—we can't just allocate space on the heap for each element individually.

We'll do the usual thing—when we run out of space, we'll call `grow`, which doubles the size of the allocation and copies the old elements over. We'd like to give a potential function argument to prove that the amortized cost of grow is $O(1)$. That is, we want a function such that amortized cost = actual cost $+\Delta\phi$ is constant, no matter what phase we're in.

So supposing we have k things in our array, and we have to call grow, we'll have $\Theta(k)$ cost (say, ck). We then need a potential function such that $\Delta\phi$ is $-ck+O(1)$. We'll make ϕ depend on the current size s (the number of elements in the array) and capacity n (note $s \leq n$).

If we let

$$\phi = c(2s - n)$$

we note that during a grow operation, we get $\Delta\phi = 2c$, or somethign? Maybe I got that wrong (go through this argument!) Note that after a grow operation, $s = k + 1$ and $n = 2k$. Note also that ϕ is nonnegative except possibly for the very beginning (which we could handle with a special case)

Suppose now we want to add a `shrink` operation: when the array gets less than a quarter full (that is, $S < n/4$), we want to halve its capacity. Note we cannot just say "halve the length of the array when it's less than half full", because this would give us a sequence of extremely expensive operations: we could go back and forth inserting and removing, right around the point where an array is full, doubling and halving the array, giving us a large fraction of operations that are very expensive.

How can we modify ϕ such that, with shrink, we still get good performance? We design a function such that, when we grow, our potential ramps up from 0 to $\Theta(n)$ (as the previous one did). However, we'll also require that when you shrink from half-full to quarter-full, we want the potential to increase to $\Theta(n)$ too (which our old one didn't do). COnsider

$$\phi = \max\{c(2s - n), c'(n - 2s)\}.$$

The first covers the case we had before; the latter covers the case where we're less than half full (the c constants are different because grow and shrink could have different costs). This function works—when we grow, we get a big change in potential; when we shrink, we get a big change in potential; in between, when incrementing or decrementing, the potential doesn't change much—it's just $O(1)$. The overall amortized cost, therefore, of these "noncritical" operations is constant. (Note that sometimes the amortized cost will end up 0 or negative; that's fine, just a side-product of the funny math going on.)

8 Splay Trees

8.1 Motivation, definition

Recall an ordinary Binary Search Tree. There, we have the invariant that, at any subtree, the value of its root is $>$ any value in its LHS subtree and $<$ any element in its RHS subtree. Doing an inorder traversal gives us the elements in sorted order.

The operations that we want are Find, Insert, and Delete. In general, these can be expensive: in the degenerate case, a BST can devolve into a linked list, and Find/Insert/Delete all take linear time. That is, these operations are all $\Omega(n)$ worst case.

We therefore consider various "balanced" binary search trees, like red-black trees or AVL trees. They're similar to BSTs, but various "rotation" operations are used to maintain $O(\log n)$ depth. A rotation is basically like picking up a tree by a node and rebalancing: swap a child and parent (and possibly move their subtrees), for example. These give find/insert/delete performance of $O(\log n)$ in the worst case.

Splay trees give $O(\log n)$ amortized time for find/insert/delete. They don't give guarantees about individual operations like the balanced trees, but we get good total performance (that is, the total cost gives the same upper bound as what you'd get with a red/black tree, but individual operations may be expensive).

Splay trees will end up being fun because the working set gets moved to near the root of the tree, so we'll get something that's actually less than the log of the number of nodes, but instead log of the working set runtime (which will almost certainly be smaller).

What is a splay tree?

- **Find**: First, do an ordinary BST find, and then do a splay operation. A **splay operations** is a sequence of **splay steps**. There are three different kinds of splay steps:

  1. A standard rotation operation. Suppose we have y the root. Suppose y has LHS child x and RHS subtree γ. Suppose x has LHS/RHS children α, β.

     We bring x up to the root position, making α its LHS subtree, y its RHS child. We make y's LHS/RHS children β, γ. Note y must be the root.

     The symmetric case does the same.

  2. A "zig-zag" step. Suppose we have some z, and x is the right child of the left child of z. Call the left child of z y.

     What we do is put x at the top, putting y as its LHS child, z as its RHS child. We move their subchildren appropriately (x's left subtree becomes y's right; x's ight subtree becomes z's left.

3. "Zig-zig". Supposeing we have x the left child of z's left child (which is y). Then we move x to the top, putting y on its right, and z on y's right (and moving the subtrees as appropriate).

A splay operation at x then repeatedly performs splay steps on x until it is at the root. If x is a child of the root, then we do the first thing. If x is at least 2 levels away from the root, then we will land in exactly one of the four above cases (the two symmetric cases of each of zig-zig and zig-zag).

Note that every one of the splay steps above retained the tree's BST property.

If we have a successful find in the BST-find, we perform splay operations to move the node to the root. If we don't (that is, x isn't in the tree), then it will have failed at some node y; we splay y up to the root.

Why do we splay y up in the case of a failure? Well, if there's a long path, then, intuitively, we don't want to keep the long path around. By splaying y to the root, we'll eventually make the tree more shallow.

- **Insert**: follow downward from the root, like you're doing a find. Once you reach a point where you would branch into an empty subtree, just insert it there. We then splay the newly inserted item to the root.

- **Delete**: This one's a bit more hairy so we don't talk about it here. Suffice it to say, we follow the same basic procedure—do a classical BST operations, and then splay something up to the root (the parent of the deleted thing, say).

8.2 Amortized analysis of find

We give a potential function argument to argue that a series of find operations is amortized logarithmic. Define the rank of x, denoted $r(x)$, as the floor of the base-2 log of the size of the subtree rooted at that node. Note that "size" here means "number of nodes", not "height".

Define the potential Φ as $c\sum_x r(x)$, summing over all nodes x (we determine what we need c to be later).

We think of the amortized cost of splay-tree find as the amortized cost of phase A + the amortized cost of phase B (the former being the BST find, the latter being the splay operation).

Assume that find reaches depth d (with $0 \leq d < n$). Then the amortized cost of phase A is the actual cost of phase 1 plus its change in potential, which is $c_0 d + \Delta\Phi$ in phase 1; since phase 1 didn't change the potential, $\Delta\Phi$ is 0, and our amortized cost is $c_0 d$ (for some constant c_0 bounding the cost that we spend in each level of the tree).

The amortized cost of phase B is $c_1 d + \Delta\Phi$ in phase 2. $c_1 d$ is the actual cost of the about-$d/2$ splay steps we need to perform. We must show that $\Delta\Phi$ in phase $2 \leq (c_2 \log n) - (c_0 + c_1)d$. That is, when we add up the amortized costs of A and B, we need to get at most some constant of $\log n$, so we need to wipe out the $c_0 d$ and $c_1 d$ terms present in the actual costs of the two steps.

We examine the sum of potential changes due to the individual splay steps performed in phase 2. For a splay step at step x, define $r'(x)$ as the rank after the step, and $r(x)$ as the rank before. We'll focus on the two zig-zag steps; we perform at most 1 of the steps that occur at the root of the tree.

How can we get at the change in potential of a zig-zag operation? Consider the third operation, zig-zig. Note that the potential doesn't change at almost every node in the tree. The only nodes where the contributions to the potential may have changed are the nodes x, y, and z involved in the zig-zag. The change in Φ is

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

Note that $r'(x)$ and $r(z)$ cancel out—they're both sitting at the root of the tree. (The same reasoning holds for zig-zag.)

We partition zig-zig and zig-zag steps into "type A" steps and "type B" steps, and then look at what their sums of potential changes looks like. Type A steps are those where $r'(x) > r(x)$. **Claim**: the total change in potential due to type A splay steps is at most $c_3 \log_2 n$ (for some c_3).

Consider again the change in potential:

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

We have that $r'(y)$ and $r'(z)$ are each at most $r'(x)$ (x is at the top of the subtree after the operation); we similarly have $r(y)$ and $r(x)$ being at most $r(z)$. So our change in potential is less than

$$\leq 2c(r'(x) - r(x)).$$

That is, the change in potential is upper-bounded by twice the increase in rank.

Note the $r(x)$ is initially nonnegative, and it can never go below $\log_2(n)$. In fact, the final rank of x is (upper-bounded by)

$$\lfloor \log_2 n \rfloor.$$

Summing over all the type A steps (the ones that increase rank), what is an upper bound on the sum? Well, it is at most $2 \log_2 n$ (that is, we've proven the claim with $c_3 = 2$). (I am missing an important statement in here somewhere.)

Intuitively, the case we're worried about is where we have a linear number of splay steps. What we've shown is that we can have at most a logarithmic number of these, since each of them causes an increase in rank.

The vast majority of splay steps will be type B, where $r'(x) = r(x)$. This happens because we take the floor of $\log_2$; unless we get over a power of 2 (which doesn't happen much), the rank doesn't change.

Claim: each type B splay step gives a potential change $\leq -c$ (for some c). This is really the heart of the argument: we're relying on the type B steps to give us a large drop in potential.

Recall the claim above that the change in potential is $\leq 2c(r'(x) - r(x))$. This isn't strong enough; in the type B case, this basically shows there is no increase in potential. What we really need to argue is that

$$r'(x) + r'(z) - r(x) - r(y) \leq -1.$$

Consider zig-zag. Suppose we have $z- > y- > x$; x ihas subtrees β, γ; y has LHS subtree α; z has RHS subtree δ. Let s be the total number of nodes in this subtree. Let t be the number of nodes in the tree rooted at x (that is, x with β and γ). We have that $t > s/2$; if it were otherwise, then the rank at x would have gone up in the element.

Now, consider the thing after the zig-zag, rooted at x. Either a or b is $< s/2$ (otherwise there would be more than s elements in the tree rooted at x). Suppose that $a < s/2$. Note either $r'(y) < r'(x)$ or $r'(z) < r'(x)$ (why?). That gives us that $r'(y) + r'(z) < 2r'(x)$ (why?). This gives us our desired result (why?).

A similar line of reasoning can be applied to zig-zig. We have $z- > y- > x$, and we switch it to $x- > y- > z$, with initially x the lhs of the lhs of z, and eventually z the rhs of the rhs of x.

Let the whole thing contain s nodes; let t be the size of x with α, β its subtrees. Now, we have the RHS-most subtree after the step (z with the otehr subtrees) have strictly less than half of s. this ends up giving us that $r'(z) < r'(x)$ (recall x ends up at the root of the resultant subtree, with LHS subtree α and rhs node y, with LHS subtree β and rhs child z.

(end proof of claim)

We DO have to say something about the last step. It's pretty simple: basically, we show we don't get a massive explosion in potential with the operation; at worst, we get at most a logarithmic term.

We have at most $\log_2 n$ Type A splay steps, out of about $d/2$ splay steps (since each gives us a bump up in rank of about 1). Thus, the number of type B splay steps has to be $\geq d/2 - \log_2 n$. Thus, the total type B drop is at least $cd/2 - c\log_2 n$; this is the claim we wanted to show above. That is, we can show that the drop in potential is $\leq c_2 \log n - (c_0 + c_1)d$. We get this by setting c_2 to $3c$, and choosing c such that $c/2 = c_0 + c_1$.

For insertion and deletion , the argument ends up being very similar. Note insertion is tricky: we potentially add another element, and so every single node on its path might have its potential increased. A naive argument will give us an insertion perhaps increasing our potential by $\Theta(n)$; however, if we notice that adding a new node can only strictly increase the rank of a logarithmic number of nodes; you wander up to the root, and by the time you reach the top, you arrive at potential at most log n (why???).

9 Augmented Binary Search Trees

9.1 Example 1: Dynamic Order Statistics

Simple technique, fairly widely applicable.

If you have a dynamic set of items drawn from some ordered universe, we can use Red-Black trees or AVL trees or something like that to get logarithmic time for insert/delete/find. However, sometimes we want one or more additional operations, typically more specialized, and we want logarithmic performance. Instead of just starting from scratch, we can frequently just augment a standard balanced search tree (e.g. red/black tree).

As a simple example, consider the problem of "dynamic order statistics". Suppose we have a dynamic collection of elements, but from time to time we'll want to run SELECT(k) queries, which returns the kth highest element in the set. If we just use a red-black tree, we can't do this efficiently—we'd have to do an in-order traversal to figure out where in the tree we want to be, and that would take linear time. However, we would like to get logarithmic time for this.

We can do so by augmenting a red/black tree in such a way (adding extra fields) that will be helpful for this. We maintain a size field at each node, which is kept equal to the number of nodes in the corresponding subtree. We'll need to work out exactly how to easily maintain this field, but it ends up being possible.

We'll have some size k_0 at the root. Say k (the query value) goes from 1 up to n. Call k_1 the size field of the root's left child (if there is no left child, then the value is 0). If the rank we want is $k_1 + 1$, then we return the value at that node. If we want a rank that's less than that, we recurse into the left tree's subtree. If we want a query of rank greater than $k_1 + 1$, we recursively search in the root's right subtree, with $k' = k - (k_1 + 1)$.

Note we never descend into both sides of any subtree; since a red-black tree has depth $O(\log n)$, we will spend logarithmic time on the query.

9.2 Augmentation Theorem

We have the following **augmentation theorem**, which is proved in CLRS (chpt 14). Suppose the auxiliary fields of a node x can be computed in constant time given the elements in x and the auxiliary+element fields of its children (that is, we can compute the auxiliary info for a node, given all the info present in its children, and the original, non-auxiliary info at that node). Then we can maintain these auxiliary fields, while achieving logarithmic performance on a red-black tree for insert/delete/find.

We give a sketch of a proof of this. We need to show that we can augment the code for insert/delete/find in such a way that we maintain logarithmic time. For find, we don't modify any auxiliary fields, so the performance is trivially the same. Now, insertion. Recall that red/black tree insertion just performs a typical BST insert (putting the new value as a leaf); in the second phase, we perform a constant number of rotations. In the first phase, we may have to percolate information up to the root, but since the tree is balanced, that will take logarithmic time. In the second phase, we may have to propagate a change up to the root again. Again, this is at worst $O(\log n)$.

This argument doesn't apply to something like AVL trees, where you might do a logarithmic number of rotations on insertion. However, we can amend the argument—you won't be doing rotations anywhere in the tree; all of them will be along a path to the root, and so you can prove that there is still only a logarithmic amount of extra work done.

The argument for deletion on red-black trees is very similar.

9.3 Example 2: Finding an overlapping interval

Suppose elements in our dynamic sets are intervals (a, b) where $a \leq b$. We want to support an operation OVERLAP(a_0, b_0) (representing the inclusive interval $[a_0, b_0]$), that either finds an overlapping interval in our dynamic set, or reports correctly that there is no such interval. How do we augment the tree?

We sort the intervals by left endpoint in the red-black tree. For example, suppose we have a root with two kids. Root has a_1, b_1; the LHS kid has a_0, b_0, and the RHS child has a_2, b_2. Suppose we have the query (a, b); what do we need to augment the tree with?

Suppose that OVERLAP(a, b) happens to overlap the root. Then we're done. If it doesn't, however, we want to query at most one of the two children. If a is between a_1 and b_1, we're done. We're left, then, with the two cases of $a < a_1$ and $a > b_1$.

Suppose $a > b_1$. If the maximal right endpoint of any interval in the LHS subtree is greater than a, then we have an overlap; if it's less than a, then every interval in the LHS subtree is guaranteed to fall completely to the left of a, and therefore there will not be any overlap.

We therefore say that our auxiliary field can be the max b-value in a subtree. If we have that, we can compare a to the max b-value of the LHS tree (assuming $a > b_1$); we can therefore recurse to the right, since we know that there is no possibility of finding an overlapping interval to the left.

Suppose now that $a < a_1$. Then we want to know what b looks like. If $b > a_1$, then we've found an overlapping interval. If, on the other hand $b < a_1$, we know that it's not possible that we get any intersection with any of the intervals in the right subtree (since the tree is sorted by a value). We can therefore recurse left.

9.4 Example 3: Finding (x, y) with $x > x_0$ and largest y

Say our elements are arbitrary (x, y) pairs in an ordered universe. Given a query x_0, we want to compute an element (x, y) with maximum y-value over all elements with x-value $\geq x_0$. For example, the x's could be release times, and the y could be values; we'd then be looking for the highest-value object after a certain release time.

Looking at it, it's not obvious if we want to sort by x or by y. It turns out you can get pretty nice solutions either way.

Solution 1: Sort by y-value. Say we have a tree with root x_1, y_1, LHS child x_0, y_0, and RHS child x_2, y_2. We're given a query with argument x.

We want to branch to the right whenever possible (we're trying to maximize y). The only time we don't want to do that is when we can prove there are no useful points to the right. When does this happen? Exactly when all of the x values in the RHS subtree are too small. We can determine this efficiently if we keep around the maximum x-value of a subtree as an auxiliary component.

When we query on x, we look at the RHS node's max x value. If it's not big enough, then we know there is no point in going that way. We can then check if the node's value itself is acceptable (in which case we return the root). Otherwise, we branch left.

Solution 2: Sort by x-value. We need to perform two passes. The first pass determines the y value; the second determines (x, y). The auxiliary field will be the max y-value.

What we want to do to find the max y value is the following. Supposing we're at the root, with a query x, we check to see if $x > x_1$. If so, then that node's acceptable, and all RHS tree nodes are acceptable. We can branch left, then, but remember the highest y value in the root or in the RHS subtree (we do that by taking the max of the auxiliary fields in the root and the RHS node). We then recurse to the LHS subtree. If $x_1 < x$, then we know that we must branch to the left—neither the root nor the LHS subtree contain any feasible nodes.

We thus come to know the y value we're seeking, and it's not too tough to figure out where to find the x value (what? how?)

9.5 Example 4: Sum all y values from (x, y) pairs with $a \leq x \leq b$

Suppose we have pairs (x, y) at nodes, and the query is (a, b). We want to determine the sum of the y-values of all elements (x, y) such that $a \leq x \leq b$. We sort the (x, y) pairs on x, breaking ties arbitrarily. Our auxiliary value is the sum of y-values in the subtree.

We give a two-pass solution. First, we traverse the find-a path (that is, run a BST find on a), computing the sum of all the y-values of all the points with x-value at least a (we can add up subtree sum values). Second, we find the sum of the y-values of all points $\geq b$. We can just subtract the latter from the former, then.

9.6 CORRECTION

In order to be able to support find efficiently, in the above, we're actually going to need to sort on lexicographic order on the (x, y) pairs (that is, break ties on the second field). Imagine the pathological case of a tree with all x values the same, for example.

10 Van Emde Boas Trees

(Chapter 20 in the 3rd edition of CLRS.)

A Van Emde Boas tree is a data structure that can substantially outperform red-black trees in some situation. Following the text, we want to support an operation MEMBER (return 1 if key in set, 0 otherwise); we won't be considering auxiliary data, but we could perhaps also store auxiliary data, and modify MEMBER to return a pointer to it. That is, we're building something that does set membership, rather than an efficient map. We also have MINIMUM; MAXIMUM; SUCCESSOR; PREDECESSOR; INSERT; DELETE.

We'll make a set of restrictive assumptions about our keys so as to be able to get sublogarithmic time on our operations. In particular, we assume the integers are drawn from a set $\{0, \ldots, u-1\}$. We'll eventually get $O(\log \log u)$ time for all operations (in fact, min and max will be $O(1)$).

First, note the simple naive bit-array version: Represent a set as a list of u bits. Then MEMBER, INSERT, DELETE are all $O(1)$. However, finding the SUCCESSOR is $O(u)$ (we could have to examine many bits before finding a 1). Similarly with PREDESSOR, MIN, and MAX.

A variation of the above would be to superimpose a binary tree structure on the bit vector: We could store a $\log u$ binary tree, where an intermediate node of value 0 means "there are no leaf nodes with 1 below this node". This gives us $O(\log u)$ SUCCESSOR, PREDECESSOR, MIN, and MAX, but the other operations won't be $O(1)$ any more. However, this doesn't really give us much over a red-black tree, which already gave us $\log n$ time, rather than $\log u$ time.

Another variation could be to use a k-ary tree, for example a $\sqrt{u}$-ary tree. This would give is OK time (figure out what it would be).

We'll use a branching factor proportional to the size of a subtree. So at the top, there are u leaves; we create $\sqrt{u}$ child nodes at the root. Each child has $\sqrt{u}$ nodes; we give each of them $u^{1/4}$ children; since there are $u^{1/2}$ of them, there will be $u^{3/4}$ nodes at that level. The depth of this tree is $O(\log \log u)$. We get this because at each level there are $u^{1/(2^i)}$ nodes (FLESH OUT THIS A BIT MORE).

10.1 Proto Van Emde Boas trees

Following CLRS, for pedagogical purposes, we first introduce a version of van Emde Boas trees that won't work out for us.

A size u node will have $\sqrt{u}$ children, each of them built in the same way (as above). However, we also have a summary of size $\sqrt{u}$ at the node—since we already have children of size $\sqrt{u}$, this won't change the size of the trees in big-O terms. The summary will look like the following. We are taking care of a subset $0, \ldots, u-1$. We imagine chunking this up into blocks of size $\sqrt{u}$. The first such one will be bits $0, \ldots, \sqrt{u}-1$ (Call this cluster 0). Cluster 1 will be the next $\sqrt{u}$ bits. We continue this way up to cluster number $\sqrt{u}-1$. Our summary will have a bit for each cluster, which is just the bitwise-or of all the bits in a cluster (that is, the semantics are "is at least one bit in the cluster a 1?").

We use the following notation from the text: when we're at a node with size u, the value x in the query (for example, MEMBER), will be a $\log u$-bit value. We use $HIGH(x)$ to refer to the most significant $(\log u)/2$ bits, and $LOW(x)$ to refer to the least significant $(\log u)/2$ bits. The HIGH bits will tell us what cluster to go into, and the LOW bits will tell us the element in the set determined by the HIGH bits.

At a node, doing an operation, we'll use the summary to skip clusters that we know have no bits in them. Supposing we're using $u = 2^{16}$, at the root node we'll have 2^8 clusters, and the summary is a representation of the integers $0, \ldots, 255$; if i is in this set, then the ith cluster has at least one number in it. Note that each of the honest-to-god children is also representing a subset of the integers 0 through 255 (that is, 0 through 2^8, representing the range of LOW bits). So the summary child looks just like the other children.

At the next level, clearly we'll look the same: each node has 2^4 children nodes and summary of 2^4 bits.

How does MIN work? The natural way is to recursively ask the summary what its MIN is; supposing that the summary's MIN is 17, then we know that we can ignore clusters 0 through 16, and we can recursively ask cluster 17 its MIN. WE have made 2 recursive calls on data structures of size $\sqrt{u}$ to get the answer of a structure of size u. This gives us the following recurrence for MIN with proto-VEB trees:

$$T(u) \leq 2T(\sqrt{u}) + O(1). \tag{20}$$

We also have the notation $INDEX(HIGH(x), LOW(x))$; when we recursively get an answer from a child, we need to concatenate two half-words together. So if we recursively get an answer at cluster k, and k returns ℓ, then our answer will actually be the concatenation of the high bits k and the lower bits ℓ. The book denotes this as $INDEX(HIGH(x), LOW(x))$ or something.

Unfortulately, unfolding recurrence (20) gives us

$$O(2^{\log \log u}) = O(\log u)$$

as its runtime (if we'd had $T(n) \leq T(\sqrt{u}) + O(1)$ instead, THEN we'd get $O(\log \log u)$). So we'll have to change our data structure to get us the desired behavior (in particular, we'll want to change the 2 coefficient, corersponding to the 2 MIN calls made, into a 1).

Similarly, what does SUCCESSOR(x) look like? First, we look in HIGH(x). If we found a successor there, we're in luck. However, if HIGH(x) is empty, then we'll have to look at the next cluster to the right that has anything in it. So we'll need to make a recursive call to the summary node, and find the MIN element in that (which we already looked at). So we'd get

$$T(n) \leq 2T(\sqrt{u}) + O(\log u) \tag{21}$$

the $O(\log u)$ is from the MIN call. This ends up being $O(\log u \log \log u)$, which is again not really what we want.

Consider now INSERT on the proto-VEB trees. What do we do? We insert LOW(x) into cluster $HIGH(x)$. If we're unlucky, the cluster HIGH(x) was empty before, and we'll need to insert into the summary node the fact that the cluster is now nonempty. That gives us

$$T(n) \leq 2T(\sqrt{u}) + O(1)$$

which is $O(\log u)$.

Note that I think MEMBER is still $O(\log \log u)$ here.

10.2 Van Emde Boas Trees: fixing things up

10.2.1 A Modification

What we need to do is patch things so that we don't need the second recursive call. What we do, intuitively, is change things so that we have two cases: an easy case and a hard case. We'll have good behavior in the easy case, and kind bad behavior in the hard case; when we need two operations, then, we'll rig things so that only one of them is "hard".

We'll maintain a MIN and MAX at each node (setting them to Nil if there are no elements). Note that, at a node, the Minimum element is not represented below.

10.2.2 An Example

As an example, suppose our set is $\{2, 3, 4, 5, 7, 14, 15\}$; then $u = 2^{2^2} = 16$. At the root, MIN=2 and MAX=15; we won't keep the 2 in its children.

The root has 4 children and the summary. The 4 clusters are $0-3, 4-7, 8-11, 12-15$ (clusters 0 through 3). Node 0 has $u = 4$; it has to represent the elements from $0-3$, except for the min (and is therefore really a representation of the set $\{3\}$. Cluster 1 is suppoesd to keep track of the ints between 4 and 7. Since it only cares about the lower-order bits, 4 gets mapped to 0; 5 to 1, and so on. So it has the set $\{4, 5, 7\}$, which is represented as $\{0, 1, 3\}$. The parent node, when it gets an answer from cluster 1, will add 4 to the result, since it knows it's cluster 1. And so on.

The summary has $\{0, 1, 3\}$, since clusters 0,1, and 3 are nonempty, while cluster 2 is empty.

We have similar tree structures for each of the above-mentioned subtrees. Each child has 2 clusters (of size 2) and 1 summary; each cluster has size 2, representing a subset $\{0, 1\}$. Recall the MIN of any node is not stored in its subtree.

10.2.3 Performance

Trivially MIN and MAX are $O(1)$ (they're both stored directly at the root). MEMBER does the following: if the value you're looking for is equal to MIN or MAX, then you know it's in the set. Otherwise, split x into HIGH(x) and LOW(x) in constant time; we ask cluster HIGH(x) if LOW(x) is a member of it; that'll give us $O(\log \log u)$ time.

A bit tricker is SUCCESSOR. We have the trivial cases with $u = 2$ (resolvable in constant time), or SUCC = MIN, in which case we can immediately return the MIN (Note that the successor is defined as the smallest value in the set that is greater than the query integer x. FLESH THIS OUT). We want to show that we can now get away with a single general recursive call.

What we do first is find MAX-LOW: find the maximimum value in cluster HIGH(x). If MAX-LOW $\neq$ NIL and $LOW(x) < MAX - LOW$, then we know that our answer must be within cluster HIGH(x), and we can just find the successor of LOW(x) inside the cluster HIGH(x), then build the answer (which is HIGH(x) concatenated with the answer we got before). If, on the other hand, either there's nothing in that cluster or the answer is in a higher cluster, then we have the same problem we had before: we look for the successor of HIGH(x) in summary (that is, find the next nonempty cluster to the right). This is $T(\sqrt{u})$. Then we get the answer as the MIN value in the successor cluster (formally, that concatenated with the successor).

WHY DON'T WE KEEP THE MIN?

For the code of INSERT and DELETE, consult chpt 20 of CLRS.

11 Union Find

Here, we'll get something growing much more slowly than even $O(\log \log u)$; in fact, we'll get an inverse Ackermann-growing function.

The problem of Union-find is the following. Suppose we want to maintain a dynamic collection of disjoint sets. Initially, we have no sets. WE have an operation called MAKESET(x), taking an elemennt from some universe, and creates the singeton set $\{x\}$ (we require that the element x not already be in some element in our collection—we require disjoint sets). Further, we have an operation UNION(x, y), which merges the sets containing x, y; that is, there should be a set in our collection with x, and one with y; after the call, we'll remove the set containing x, remove the set containing y, and then insert the union of those two sets. Finally, we have FIND(x), which returns the "name" of the set containing x (we require that x be an element in some set in the data structure already). It doesn't really matter what this "name" is; we only require that x, y are in the same set iff FIND(x) = FIND(y).

Tarjan discovered the following union-find data structure, which uses trees to represent each of the sets. Each element of a set is a node in a tree; each element has a pointer pointing to its parent (there are only upward-pointing arrows in the tree). For any node, we simply follow the upwards parent-pointers until we get to the root. Note there are many possible representations of a set; all we require is that it be a tree. The naming convention will be to return the root element as the name of a set.

Note that on a find call, we're given directly the node in the tree; we don't have to look around in different trees to find an element.

If we want to do a UNION on two elements in the same set, we don't need to do anything. If they're in different trees, we do the following: make one of the root nodes a child of the root of the other one. Following CLRS, define LINK(T, T') as an operation taking as arguments two roots of trees; this operation makes one of them point to the other (that is, one of the roots is a child of the other root).

Given what we've defined, we can imagine degenerate cases with bad behavior. In order to get around these, we have two tricks.

First, we have **path compression**. Whenever we do a FIND, we traverse the "find path"—the path from the found node to the root. With path compression, we retraverse the path a second time, and make everything point directly to the root. That is, if we traverse a path of length 100 to get to the root, we make 99 elements that didn't used to point to the root point directly at it. Subsequently, FIND on any of those elements will be very cheap (assuming we haven't done too many intermediate UNION operations). This gives us pretty good performance by itself.

Second, we have **union by rank**. With this operation, we'll be able to almost get amortized $O(m)$ time of any sequence of m operations. In fact, what we'll get instead is m operations taking $O(m\alpha(m))$ time, with m the number of MAKESET calls, and α the inverse Ackermann function; however, the latter is very very slow.

Anyways, back to **Union by Rank**: if you have a huge structure and a tiny one, it'll be better to make the tiny structure a child of the large one. Intuitively, if we make the large structure a child of the small one, then we would increment the depth of a whole bunch of nodes; if we make the small one a child of the large one, we only increment the depth of a small number of nodes.

Instead of making this decision by size, we'll make it by **rank**, which is a proxy to size, which will end up giving us the same optimal bounds. Rank is a nonnegative integer. When we do MAKESET, initialize the element's rank to 0. A FIND has no impact on rank; when we call LINK(x, y); if x has rank r and y has rank r', and assuming wlog that $r \geq r'$, then we make y a child of x. If $r > r'$, then there is no change in the rank of x; if $r = r'$, then we increment the rank of x. That is, regardless of the structure of the tree, if we link two trees with the same rank, then the rank is the previous rank plus 1; otherwise, the rank is what it was beforehand.

A side note: you can show pretty easily that the $r(x) \leq \log_2 n$, with n the size of the subtree rooted at x; further, the depth of the tree rooted at x is at most the rank of x. This can be shown by induction on the tree size.

Now, the Ackermann function will look as follows:

$$A_0(j) = j + 1 \tag{22}$$

$$A_1(j) = 2j + 1 \tag{23}$$

$$A_2(j) = 2^{j+1}(j + 1) - 1 \tag{24}$$

$$A_3(j) = 2!j \tag{25}$$

where $2!j$ is a tower of repeatedly exponentiated 2's (that's notation people use, right?). I can't type A_4, but it's a tower of 2's, whose height is a tower of 2's, whose height is a tower of 2's, and so on. The functional inverse of doubling is halving, and the functional inverse of exponentiation is log, and the functional inverse of the tower is $\log^\star$, which is something or other (I dunno).

Define $\alpha(n)$ as the minumum k such that $A_k(1) \geq n$. Note the real function used in Tarjan's paper is a bit more complicated (it's the Ackermann function with two arguments).

Formally,

$$A_0(j) = j + 1 \tag{26}$$

$$A_k(j) = A_{k-1}^{(j+1)}(j) \tag{27}$$

where $A_k^{(j)}$ is A_k iterated j times.

We'll see that the running time of m operations is $O(m\alpha(n))$, with n the number of MAKESETs. Note LINK is constant-time worst-case, MAKESET is constant time. The difficulty is in FIND—we can have a long path to the root, but we'll want to show that this doesn't happen too often.

We'll use a cleverly-defined potential function to argue that the amortized cost of each FIND is $O(\alpha(n))$. NOte that LINK will still have amortized $O(\alpha(n))$, even though it's worst-case cosntant time (because it can potentially increase the potential). When we do a FIND, we'll want to argue that the potential function drops in such a way that it basically entirely offsets the cost of traversing the path to the root. Since we pay 1 unit of actual cost going up a step in the path, we'll need our potential go down 1; there will be $\alpha(n)$ nodes in the path in which we can't make that argument, so we'll have to pay the unit cost at each of those nodes.

The overall potential Φ will be

$$\Phi = \sum_x \phi(x)$$

with x ranging over all nodes in the data structure, and $\phi(x)$ being

$$\phi(x) = r(x)(\alpha(x) - LEVEL(x)) - ITER(x)$$

where $LEVEL(x)$ is the following: for a node x, we look at the rank of the parent of x, defined as r'; if r' is much bigger than x, we'll ahve a high LEVEL, otherwise we'll have a low LEVEL (intuitively, it will encode "how much higher a parent's rank is than yours"). Precisely, defining $r'(x)$ as the rank of x's parent, $LEVEL(x)$ is the smallest k such that $A_k(r(x)) \geq r'(x)$. Note that for any non-root x, we have that $r(x) < r(p(x))$ (with $p(x)$ the parent of x) (it is somewhat easy to convince yourself of this). So the question is, roughly, "how much higher" is your parent's rank than yours? Note that LEVEL will almost always be really small.

Below, we abbreviate $LEVEL(x)$ as $\ell(x)$ and $ITER(x)$ as $i(x)$.

LEVEL is a primary measure of how much higher a parent's rank is than yours. ITER is a secondary one. $ITER(x)$ is the largest k such that

$$A_{\ell(x)}^{(k)}(r(x)) \leq r(\phi(x))$$

with $A_{\ell(x)}^{(k)}$ denoting k successive applications of the Ackermann function $A_{\ell(x)}$ (and with $\ell(x)$ the LEVEL function). $\phi(x)$ is

$$r(x)(\alpha(n) - \ell(x)) - i(x)$$

for a nonroot x with $r(x) > 0$; and $r(x)\alpha(n)$ otherwise.

Note some properties of $\ell(x)$: first, $0 \leq \ell(x) < \alpha(n)$. I'm not quite exactly sure why this is, but I think it's basically just an application of the definition of LEVEL. SEcond, the level of a node cannot decrease. We might get a new parent after path compression; however, since the rank of the parent of x cannot decrease under any operation, and the rank of a parent is always larger than its kids, then the rank of a parent will only stay the same or increase.

Note the following tow properties about $i(x)$. First, while $\ell(x)$ does not change, $i(x)$ cannot decrease either. This follows from a similar line of reasoning for why $\ell(x)$ cannot increase, I think (but I'm not sure). Second, $1 \leq i(x) \leq r(x)$. To show thise, we use contradiction. Suppose $i(x) \geq r(x) + 1$. Note that

$$A_{\ell(x)}^{r(x)+1}(r(x)) \leq A_{\ell(x)}^{i(x)}(r(x)) \leq r(p(x))$$

for some reason; further, the LHS-most term in athe above is equal to

$$A_{\ell(x)+1}(r(x))$$

This is a contradiction for some reason I don't understand at all.

Heading back, note that $\phi(x) \geq 0$ for every x (why?), so our potential Φ is positive; further, $\Phi = 0$ initially.

Let's look now at the amortized cost of a FIND operation. WE could have some sort of really long FIND path (linear in m, say); we'll want to show that the amortized cost is $O(\alpha(n))$. We'll need to offset that cost with a corresponding drop in potential. Why would this happen? Intuitively, for a whole lot of nodes, the comparative rank of the nodes' new parents will be higher than their old parents'.

WE get through some convoluted argument that, for any node x, the drop in potential given by that node is at least 1. I couldn't follow the argument because I am sick and can't focus on anything. It seems like the argument is probably laid out in CLRS though.

At the end of the day, the total amortized cost of a FIND is O(number of nodes x on the FIND path for which $\phi(x)$ is unchanged), that is, for which $\ell(x)$ and $i(x)$ are unchanged; those nodes contribute 0 to the total cost, whereas the other nodes cancel out the unit amount of work needed to visit/move that node.

How on earth can we upper-bound that number? Well, suppose we have a bunch of level 7 nodes, say. Then for every node except possibly the topmost one, we can be sure that the level will drop. Why? well $\phi(x) \geq A_7^{(12)}(r(x))$ but less than $A_7^{(13)}(r(x))$. Then consider every level 7 node except for the topmost one; each of these leapfrogs someone of the same level, and something interesteing happens there. or whatever, jesus i have no idea.

More concretely, suppose we have three rank-7 nodes a, b, c, with a the furthest from the root, b the closest; they're along a path. Now, calling y the new root, we have

$$\begin{aligned} r(y) &\geq r(c) \\ &\geq A_7(r(b)) \\ &\geq A_7(A_7^{(12)}(r(x))) \\ &= A_7^{(13)}(r(x)) \end{aligned}$$

and I have no idea what this means, but apparently it gives us some insight as to why only the topmost level-7 node might be problematic. This, further, supplies the insight for why the prose-language expression inside the big-O above is $\leq \alpha(n)$.

We need also think about the amortized cost of a LINK operation. Supposing we link x and y with $r(x) \leq r(y)$, then $\phi(x)$ cannot increase, but $\phi(y)$ could increase, I think by $\alpha(n)$ for some reason. Oh, I think it's entirely due to a possible increase in potential, and has nothing to do with cost. This still gives us what we want, though.

12 Basic Graph Algorithms

12.1 DFS

Depth-first search and Breadth-first search are defined for both directed and undirected cases. The directed case is just a more general case: replace each undirected arc with two directed ones. We consider directed search then.

So DFS is $O(|V| + |E|)$ time to visit all nodes and vertices; in this context, that's a linear-time algorithm, and the best we could hope for. To choose a particular DFS, we must choose an arbitrary ordering of vertices and an arbitrary ordering of the edges leaving each vertex.

In addition to visiting everything, a DFS classifies each edge as a tree, back, cross, or forward edge. Note that these aren't inherent properties of edges; different edges will be different types depending on the ordering used to do the DFS. These are useful because, for example, a digraph G contains a cycle iff any DFS of G contains a back edge.

The tree edges yield arborescences: tree-like structures where the edges are all directed away from the root. Suppose we start a DFS at some node. If we haven't seen a node, we'll make a recursive call to DFS, which must complete before we reach the rest of a tree. It's somewhat easy to see that the tree edges here to form a tree structure; if there is a node that breaks the tree structure, then it won't be a tree. Note that not all nodes will necessarily be encountered on a DFS.

For a vertex u, define $d(u)$ as the discovery time: the time u was first encountered on the search; define $f(u)$ to be its finish time, the time at which we realize we have discovered all outgoing vertices of a vertex (and DFS returns, I guess?). We have the following coloring scheme: until a vertex is discovered, we refer to it as being **white**. Once it's discovered, it turns **grey**. After the finish time, it turns **black**. Initially all vertices are white.

We have the **white path lemma**, which states that if there is a white path from u to v when u is discovered, then

$$d(u) < d(v) < f(v) < f(u).$$

We also have the fact that the discovery/finish intervals for any intervals "nest": that is, they can be disjoint, or the d/f interval for one vertex can be entirely inside that of another vertex; hoever we can't have overlapping nonnesting intervals—essentially, we have a call stack of vertices being visited, and this is a property of stacks.

So the white path lemma is proved pretty straightforwardly from this latter fact. We can use induction on the length of the path between u and v.

Another useful property is the following: the d/f times of v nest inside those of u iff v is a proper descendent of u. This is pretty obvious and pretty easy to prove.

Note, being a bit more formal about the edge classifications, suppose we are looking at an edge from u to v; u will be gray.

- If v is white, then (u, v) is a tree edge.
- If v is gray, then (u, v) is a back edge.
- If v is black, then (u, v) is a forward or cross edge. It's a forward edge if $d(u) < d(v) < f(v) < f(u)$, and is a cross edge otherwise. This uses the fact that v is a descendent of u in the forest defined by tree edges iff $d(u) < d(v) < f(v) < f(u)$; that is, it's a forward edge iff v is a descendent of u.

  Note that cross edges either go from a node to a non-descendent node in the same arborescence, or from one of the trees to a different previously-discovered tree.

We give four little applications of DFSs:

- **Strong connectivity:** To determine if a graph is strongly connected, we can do the following: pick an arbitrary u, do a DFS; if everything is blackened, then you can get from anywhere to u; the second step is to reverse the graph and do the same; if you've shown you can get to anywhere from u and from anywhere to u, then your graph is strongly connected.
- **Directed cycle detection:** Second, we can do cycle recognition easily. If any directed search gives at least one back edge, then you have a cycle. That is, no matter what search you do, you get at least one back edge.

We prove both directions. First, suppose you have a back edge, we show a cycle. The gray node we're operating on currently has a greater discovery time than any of the other gray nodes; this follows pretty straightforwardly from the stack-based discipline of the recursive algorithm. If we have a back edge from u to v, then $d(v) < d(u)$, since we're using a stack. Since neither of them is finished (they're gray), we have from the nesting property that $d(v) < d(u) < f(u) < f(v)$; that is, u is a descendent of v, and so we get a cycle.

Second, supposing we have a cycle; we need to show that we get a back edge under any DFS. Suppose we have a cycle. Every vertex will eventually get discovered in the DFS. Take our cycle C; say we have u and predecessor v; we have from the white path lemma that every path on the cycle will be discovered. Suppose we start at u and end up at v, u's parent; since these things nest, v must complete before u (or something); since they're both gray, this will be a back edge (THIS ARGUMENT ISN'T QUITE RIGHT).

- **Topological sort of a DAG:** Suppose we know we have a DAG. Our goal is to order the vertices so that all the edges go from left to right (that is, if $(u, v) \in E$, then $u \leq v$ in the ordering). This is easy: run a DFS, and sort the nodes in descending order of finish time. So we run a DFS; as we turn vertices black, just stick them at the head of a list. This results in a topological sort.

  In particular, we need to show that there is no way an edge can go from right to left. Suppose for contradiction that there's some edge (u, v) going from right to left; that is $u > v$; that is to say, $f(v) > f(u)$ (recall we're sorting by DESCENDING finish time).

  So when the edge (u, v) is processed, u is gray; and v is one of the three colors. If v is gray, then it's a back edge; since that implies a cycle and we assume a DAG, that can't be. If it is black, then this implies $f(v) < f(u)$, which contradicts the ordering we imposed on the vertices (namely, that $f(v) > f(u)$). Finally, supposing v is white, by nesting, we must have that v finishes before u does, which again contradicts the ordering we defined on the vertices.

- **Computing strongly connected components of a graph:** Doing a DFS gives us a collection of arborescences spanning the entirety of the graph G. Note that the cross-edges all have to go from "right to left" from later-discovered trees to earlier-discovered trees (otherwise, then the rightmost tree would have been a part of the leftmost tree).

  A strongly connected component with a component v contains just those vertices that reach v and can be reached by v. Suppose that v is the root of some tree T_k discovered by DFS; the strongly connected component with v must be entirely within T_k, since no vertex in an earlier tree can reach v. Now, we've already determined that v can reach everything in

T_k; we now have to determine which vertices in T_k can reach v. Consider reversing the edges and running a DFS. Note that if we do DFS in the reversal, we won't leave tree T_k; since the only edges leaving the tree go to previous trees, those will be reversed now. So the reversal of graph T_k will only lead us to nodes in T_k if we start at v.

Note also that the strongly connected component with v will consist of the "top" of v; that is, it's not possible to have a vertex u in v's strongly connected component that doesn't have a path from v to u in the tree (thik about this a bit more).

So we remove v's strongly connected component, and we're left with a bunch of little trees that were subtrees of T_k; what's left looks a lot like what we had before—we replace T_k with 0 or more trees that have an ordering on them, and may have cross-edges going from right to left. We then recurse into the rightmost created subtree and do the same.

All we need to do, then, to get the strongly connected components of a graph G is to (1) run DFS on G; (2) run DFS on the reversal of G, with the outermost loop in descending order of finish times from the first DFS (That is, start at the rightmost arborescence discovered by the DFS; always start at the tree whose root has the highest finish time).

12.2 BFS

We can use breadth-first-search to do **SSSP**–single-source shortest path—in an unweighted digraph. Suppose we have a single source s. We can find all things at distance 1 from s by looking at all of s's children; we can find all things at distance 2 by looking at their kids, etc. This is exactly what a BFS gives us.

Recall that BFS simply uses a queue rather than a stack to keep track of where we are. So we put all of s's kids into a queue, and blacken s. At that point, our queue contains all and only those things 1 elem away from s. Looking at s's first child x, we look at all its kids, and add white ones to the queue. Once we've processed all nodes of distance 1 from s in the queue, we will have put all and only those distance-2 vertices into the queue, and so on.

BFS is of course $O(|V| + |E|)$; enqueueing and dequeueing is constant time.

12.3 Weighted SSSP, APSP

Recall Dijkstra's algorithm, which solves SSSP for weighted graphs. Assuming edge weight ≥ 0, we can use, e.g., a Fibonacci heap to get $O(|E| + |V| \log |V|)$ for SSSP. (Note that if we use a simple binary heap, we get $O(|E| \log |V|)$, which is slightly worse.)

Consider APSP, all-pairs shortest path. Naively, we can run Dijkstra's algorithm for every vertex, which will solve APSP in $O(|V||E| + |V|^2 \log |V|)$.

However, the above works only for graphs with nonnegative edge weights. If we allow arbitrary edge weights, assuming no negative-weight cycles, we don't get very good performance. We have, for example, Bellman-Ford, but that is $O(|E| \cdot |V|)$, which is decidedly nonlinear.

Consider APSP for a graph with no negative-weight cycles (but with arbitrary edge weights). That is, we want the SP for every pair of nodes in the graph. Johnson's algorithm gives the same bound as using Dijkstra's algorithm $|V|$ times, which is cool (that is, recall, $O(|V||E| + |V|^2 \log |V|)$). Basically, it uses one round of Bellman-Ford to convert to a graph with nonnegative edge weights that retains the shortest paths, and then runs Dijkstra's $|V|$ times.

A side note, this general technique of first converting to graphs with only nonnegative edge weights and then using an algorithm that works for nonnegative graphs is somewhat common.

Note that Dijkstra's algorithm is kind of a modification of BFS. We follow basically the same strategy as in a BFS—examine the closest things first, then the next closest, and so on. First, we find everything less than or equal to some distance t; everything else that we haven't found has distance $\geq t$.

Call X the set of things we've looked at, Y the set of things we haven't. For every unexamined vertex y, we maintaing a value

$$d'(y) = \min_{x \in X} d(x) + w(x, y).$$

This is an obvious upper bound on the shortest path distance of y; if we can get to x in $d(x)$ cost, then we know we can get from x to y in cost $w(x, y)$, and the cost of y is $d'(y)$; there might be a better way, but we can upper bound it thusly.

In Dijkstra's algorithm, we store everything in a heap based on this upper-bound value $d'(y)$. What we do iteratively is remove a vertex $y^* \in Y$ minimizing $d'(y^*)$ (there may be multiple such y^*'s, of course). We have that $d'(y^*) = d(y^*)$; that is, the upper bound that d' gives us is tight.

With a Fibonacci min-heap, we can get decreaseKey in amortized constant time (wtf is this); anyways, we don't get that with a regular Binary Heap.

12.4 Bellman-Ford

Bellman-Ford solves SSSP for graphs with arbitrary weight edges and no neg-weight cycles. How does Bellman-Ford work? For all u, let d_u denote the shortest path distance from the source to u. The key observation is that if there is an edge (u, v) with weight $w(u, v)$, then we have

$$d_v \leq d_u + w(u, v)$$

That is, we can definitely get from the source to v in by going through u and then from v; there may be a shorter path.

The algorithm works iteratively: we compute a sequence of values

$$d_u^{(0)}, d_u^{(1)}, \ldots, d_u^{(|V|-1)}$$

for each vertex u; we set $d_u^{(0)}$ to ∞ if u is not the source; at the source, we set the distance to 0.

At each step, then, we perform the following relaxation operation for every edge (u, v):

$$d_v^{(i+1)} \leftarrow \min\left\{d_v^{(i)}, d_u^{(i)} + w(u,v)\right\}.$$

The key property here is the following claim: for every i, $d_u^{(i)}$ is the length of a shortest path from the source to u with at most i edges.

We can prove the claim by induction on i. A brief proof sketch follows. Suppose the claim is true for i; we want to prove it for $i+1$. Let P be a shortest path from s to u with at most $i+1$ edges. We must show that $d_u^{(i+1)}$ is the length of the path P. Let v the predecessor to u on the path P; then P', the path from s to v, must be a shortest path to v with at most i edges. The Induction Hypothesis entails that $d_v^{(i)}$ is the length of P'. By the definition of $d_v^{(i+1)}$, we'll get the desired result.

This algorithm is $O(|V||E|)$.

A tiny remark is that it's not necessary to maintain separate variables $d_u^{(i)}$ for each stage; we can just use d_u variables. The wording of our claim changes slightly, but it's not super-important.

12.5 Johnson's Algorithm for APSP

Floyd-Warshall gives an $O(|V|^3)$ algorithm for APSP. Johnson's algorithm gives us better behavior.

Suppose we have a graph $G = (V, E)$ with no negative-weight cycles (but which has arbitrary edge weights). We modify G to get G' by adding an additional node z and edges of weight 0 going to every node in V. For every $v \in V$, let d_v be the weight of a shortest path from z to v in G. Note d_v is possibly negative (that is, isn't just trivially 0). We compute the d_v values using Bellman-Ford, which is $O(|V||E|)$.

Now, we define $\widehat{G}$ as G (that is, it doesn't have z), but with $w(u, v)$ replaced by

$$\widehat{w}(u,v) = w(u,v) + d_u - d_v.$$

We, crucially, have the property that $\forall u, v[\widehat{w}(u,v) \geq 0]$. Consider arbitrary u, v connected in G with $w(u, v)$. We have

$$0 \leq d_u + w(u,v) - d_v$$

(why?! We get it I think because $d_v \leq d_u + w(u, v)$.)

So we have $\widehat{G}$ with nonnegative edge weights; we can use Dijkstra's algorithm $|V|$ times to compute APSP in $\widehat{G}$ in $O(|V||E| + |V|^2 \log |V|)$ time, which is fine.

However, does this tell us anything about the answer to APSP in the original graph G?

Consider some path from $u = v_0$ to $v = v_k$, which goes through $v_1, \ldots, v_{k-1}$. We ask about a path in G, however, there will also be a path in $\widehat{G}$, just with different edge weights. The weight of P in G is

$$W \equiv \sum_{0 \le i < k} w(v_i, v_{i+1}).$$

Now, the weight of P in $\widehat{G}$ is

$$\begin{aligned}&\sum_{0 \le i < k} w(v_i, v_{i+1}) + d_{v_i} - d_{v_{i+1}} \\ =&W + d_{v_0} + d_{v_k} \\ =&W + d_u + d_v\end{aligned}$$

where we have the second line because the first is a telescoping sum.

Since we add the same thing to every path, then a shortest path from u to v in $\widehat{G}$ is also a shortest path in G.

If our graph is not dense, then this is an asymptotic improvement over Floyd-Warshall (if on the other hand $|E| = |V|^2$, then the graph has the same asymptotic complexity).

13 Maximum Flow

We are given a directed graph $G = (V, E)$. Each edge (u, v) has a non-negative capacity $c(u, v)$ (we typically consider integral capacities). We have two distinguished vertices s and t, the source and sink, respectively. We are interested in, intuitively, maximizing the flow from s to t (think analogically about pipes). We have the following two constraints. First, defining $f(u, v)$ as the amount of flow we assign from u to v (assuming u and v have an edge connecting them), $0 \le f(u, v) \le c(u, v)$; that is, we can't assign more flow than there is capacity. Second, flow conservation must hold for all nodes not in $\{s, t\}$; that is,

$$\forall v \in V \setminus \{s, t\} \left[\sum_u f(u, v) = \sum_u f(v, u)\right]$$

where we use the convention that if $(u, v) \notin E$, then $c(u, v) = 0$; this allows us to write expressions like $\sum_u$ without considering which edges are in the graph. The above equation, in other words, states that the flow into v is equal to the flow out of v.

Our goal, then, is to maximize the net flow out of s:

$$\sum_f f(s, v) - \sum_v f(v, s)$$

which is the flow out of s minus the flow into s.

We have the notion of a **cut**: a cut (S, T) is a partition of V into two sets S and T so that $s \in S$ and $t \in T$. Note there are $2^{|V|-2}$ cuts (for every vertex in $V \setminus \{s, t\}$, we can put it in either of the two sets).

Note also the value of flow across a cut (S, T) is

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

Note that the net flow out of s is the value of f across the cut

$$(\{s\}, V \setminus \{s\}) \,.$$

Note that for a fixed flow f, the value of the flow is the same across all cuts (that is, for any cut, we'll get the same value of flow, so our maximization problem is perfectly well-defined).

The proof of the above looks something like following. Fix a cut (S, T); we'll argue that the value of the flow across this cut is equal to the value of the flow across the special cut $(\{s\}, V \setminus \{s\})$. We rewrite the equation giving the value of flow across a cut (S, T) to get

$$\sum_{u \in S} \sum_{v \in S} f(u, v) - f(v, u) = 0$$

We have that the flow across (S, T) is equal to itself plus the above, which is equal to

$$\sum_{u \in S} \sum_{v \in V} f(u, v) - f(v, u)$$

Augh, what? Now, the net flow out of u is equal to 0 unless $u = s$, so we have the above equal to

$$\sum_{v \in S} f(s, u) - f(v, s)$$

since all but one term in the outermost sum is equal to 0.

We are therefore free to refer to the "value of f", regardless of the cut; that will be equal to the value of the flow across any cut.

For a cut (S, T), we define the **capacity of a cut** to be equal to

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

Importantly, we are not taking into account the edges going back from T to S. Intuitively, this is a naive upper-bound on the value of the flow across the cut (S, T): hypothetically maximizing flow out of S and into T (and ignoring flow back from T to S) gives us an upper bound on the max flow.

13.1 Ford-Fulkerson algorithm, Max-flow Min-cut theorem

Now, a **minimum cut** is defined to be a cut of minimum capacity.

We have the **Max-Flow Min-Cut Theorem**, which states that the value of a max flow is equal to the capacity of a min cut. We prove this by proving the two inequalities.

First, let (S, T) be a min-cut. We have that the capacity

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

is an upper-bound on the flow across (S, T):

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

which tells us that the value of a max flow $\leq$ the capacity of a min cut.

In the other direction, we want to show that the capacity of a min-cut is at most the value of a max-flow. To show this, we need another defintion. Define a **residual network** G_f of a given flow-network G with respect to a given flow f.

The idea of this network is to model the leftover "unused" capacity in the network G. That is, for nodes u, v, we have flow values $f(u, v)$ and $f(v, u)$, and $c(u, v)$ and $c(v, u)$. Considering only these values, what is a naive upper-bound on the amount of increase in the net flow being pushed from u to v along these edges? Well certainly, we can increase flow from u to v by

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

which uses up unused capacity. We can also eliminate any flow going from v to u, which would increase the net flow; the most we can do so is

$$f(v, u).$$

So the most we can hope to increase the net flow from u to v, ignoring everything else in the network, is

$$c(u, v) - f(u, v) + f(v, u) \equiv c_f(u, v)$$

So the residual network will be a network with capacity $c_f(u, v)$ going between u and v; note that we get this to be a valid edge capacity through a simple argument of some sort involving the definition of $c_f(u, v)$.

For every pair of vertices u and v, we go through this calculation. It is convenient to remove from G_f any edges (u, v) with zero residual capacity (with $c_f(u, v) = 0$). This will be useful because we'll be interested in determining the presence of directed paths from s to t in the residual network.

Suppose there is a path P from s to t in G_f. Let's say there are 4 edges on P, with weights 5, 3, 12, and 3. We claim that f is not a max-flow; if we were to augment f by pushing an additional 3 units of flow along the path. More generally, we can find a flow f' with value equal to value of $f + 3$ along the path, and unchanged everywhere else. We do this by adjusting the flow between two nodes u, v on the path either increasing $f(u, v)$ or backing off the flow on $f(v, u)$; to do that, we will have increased the net flow by 3, but may violate the conservation principle. However, we also push an additional 3 units of flow to the next rightmost node, which offsets the change we made before. By following this logic all the way along to t, we will have routed 3 units more flow from s to t. A path of this sort is called an **augmenting path**.

An algorithm emerges from this: start with an all-zeroes flow; repeatedly construct the residual network, using a DFS to check the existence of an (s, t) path, and repeat. This is the **Ford-Fulkerson algorithm**.

Suppose no $s - t$ path exists in G_f. Let S be the set of nodes reachable from s in G_f, and let $T = V \setminus S$. Let's return to the other half of the max-flow min-cut theorem that we haven't showed yet. We cannot get from anything in S to anything in T; for every $u \in S$ and $v \in T$, the residual capacity is 0; if it were otherwise, then the edge would be present. The only way a residual capacity can be 0 is if $c(u, v) - f(u, v) = 0$ and $f(v, u) = 0$. That is, there is nothing flowing back from $f(v, u)$, and $f(u, v)$ is at its capacity. Following this logic across the cut tells us that the value of f is equal to the capacity of (S, T).

The max-flow is greater than or equal to f, which is the capacity of cut (S, T), which is greater than or equal to the capacity of a min-cut. (Since (S, T) is an arbitrary cut, the min-cut will be less). We have therefore established the second inequality, and we've proven the max-flow theorem.

Given that we've proven the theorem, we actually have tight bounds, and f is a max-flow, and (S, T) is a min-cut. In other words, our algorithm gives us max-flow; additionally, it gives us a min-cut.

Note that we called Ford-Fulkerson an algorithm, but it shouldn't really properly be treated as one: we don't specify how to pick the augmenting path from s to t; in the worst case, the time complexity is pretty bad, if you make bad choices about which augmenting paths to choose. In particular, it can be linear in the sum of the max-flow; that is to say, the algorithm is pseudopolynomial—we'd like it to be independent of the magnitude of the flow, assuming it fits in a word or whatever, but it isn't.

More concretely, this is

$$O(|f^*||E|)$$

with f^* the magnitude of the max-flow, and $|E|$ the number of edges.

If we modify this to use a BFS somehow (I'm not sure how, look this up), we get what's called the **Edmonds-Karp algorithm**. This ends up being $O(|V||E|^2)$.

13.2 A lemma about Edmonds-Karp

Suppose $d_f(s, v)$ is the shortest path length from s to v in G_f. We show the following fact about Edmonds-Karp:

For any vertex $v \in V \setminus \{s\}$, $d_f(s, v)$ does not decrease.

We use induction. Suppose we have two steps G_f and $G_{f'}$, before and after a step (I think). WE must show $d(f(s, v) \leq d_{f'}(s, v) \equiv k$.

Supposing $k = 0$, this is trivially true, since $d_f(s, v)$ is always nonnegative (I think).

For the inductive step, We consider $d_{f'}(s,v) = k+1$, assuming it holds for k. WE have a path from s to some u, and then an edge from u to v. WE'll have that

$$\begin{aligned} d_{f'}(s,v) &= d_{f'}(s,u) + 1 & (28) \\ &\geq d_f(s,v) + 1 & (29) \end{aligned}$$

the last step is by IH. There are two cases: either the last edge (u,v) is in the resididual graph, or it isn't. If $(u,v) \in E_f$, then we have

$$d_f(s,v) \leq d_f(s,u) + 1$$

I have no idea why this is true.

The other case is where $(u,v) \notin E_f$. This can only happen if we cancel some flow going through a different edge (what? why?). In other words, we pushed flow from v to u to obtain f' from f. That is, the reverse edge (v,u) must appear in G_f. Then we have that

$$d_f(s,u) = d_f(s,v) + 1$$

for some reason. Note that here we use Edmonds-Karp somehow—we assume that the path we choose is the shortest path.

We thus have

$$d_{f'}(s,v) \geq d_f(s,v) + 2$$

by putting these two parts together.

13.3 Runtime of Edmonds-Karp

WE can safely assume the number of edges is at least $|V|$; if $|E| < |V|$, then we can throw out some vertices and our max-flow will be the same. So we'll show that Edmonds-Karp is $O(|E||E||V|)$. We use the previous lemma.

Whenever we augment the flow along a path, at least one of the edges will become saturated—we augment by the min value along an augmenting path. This edge (the one that gets saturated and disappears in the next residual graph) is known as a **critical edge**.

We will argue that each edge can be a critical edge at most $|V|/2$ times. This will give us the result—each time we do an augmentatoin, at least one edge becomes critical, so $|E||V|/2$ becomes an upper-bound on the number of augmentations we perform, and each of those is $O(|E|)$ (note BFS is actually $O(|V| + |E|)$, but recall that we assume $|E| \geq |V|$).

Suppose (u,v) is critical during an augmentation. Then the augmentation path has some $s-u$ path and some $v-t$ path, with (u,v) on the path. Note we have

$$d_f(s,v) = d_f(s,u) + 1$$

since (u,v) is on the path.

In the next residual graph, (u, v) is not an edge, since it's critical; however, (v, u) will appear. If (u, v) appears in some future residual residual network $G_{f''}$, then we will have had to push flow from v to u in the residual previous to $G_{f''}$. The path at that point will have to have an $s - v$ path, have (v, u), and then a $u - t$ path. Further,

$$d_{f'}(s, v) = d_{f'}(s, v) + 1$$

where f' here ISN'T the f' it was before, but instead the flow of the residual network that's the immediately predecessor to f'', I think.

By our lemma, however, this tells us that

$$d_{f'}(s, u) \geq d_f(s, u) + 2.$$

In other words, if the edge (u, v) appears again, then the distance of the critical path will have increased by at least two. Since the distance of any node is $|V|$, then something can be a critical path at most $|V|/2$ times.

So since there are $|E|$ edges, each of which is a critial path at most $|V|/2$ times, there are at most $O(|E||V|)$ steps in the algorithm.

In other words, we've shown that Edmonds-Karp is a true polynomial algorithm, as opposed to Ford-Fulkerson, which is pseudopolynomial.

13.4 Generic Push-Relabel algorithm

We give a "generic" push-relabel algorithm, which will achieve an $O(|V|^2|E|)$ uppser bound for max-flow (compared to Edmonds-Karp, which was $O(|V||E|^2)$). This is an improvement, as $|V|$ is better than $|E|$ in the max-flow setting.

Later, we'll further improve this to $O(|V|^3)$. The generic algorithm is nondeterministic (there will be a step of the form "choose some node from a list"). Later we'll give a deterministic one, implementing the nondeterministic one we present here, that will be $O(|V|^3)$.

We'll build **preflows**, rather than flows. They respect capacity constraints (no edge can have a flow greater than its capacity, and every flow is nonnegative); however, instead of flow conservation, we'll require that the net flow into any $u \in V \setminus \{s\}$ to be nonnegative (so we can have more flow in than out, but not vice versa).

For a vertex u, define

$$e(u) = \sum_v f(v, u) - \sum_v f(u, v) \geq 0.$$

This is the excess flow: how much more flow comes in than is going out. It's will be nonnegative: at no point in our algorithm can it go below 0. A vertex is said to be **overflowing** if $u \in V \setminus \{s, t\}$ and $e(u) > 0$.

We'll define residual networks as we did above: when defining $c_f(u, v)$, the residual capacity under f from u to v, we construct it locally: we look only at $f(u, v), f(v, u), c(u, v)$, and $c(v, u)$.

We don't just maintain a preflow; we maintain a preflow and a **height function**, mapping each node to a nonnegative integer. Now, at all points, $h(s) = |V|$, $h(t) = 0$. For $u \in V \setminus \{s, t\}$, $h(u)$ will be initially 0; it may increase at times from relabel operations (defined below).

So (f, h) capture the state of the algorithm; call one such (f, h) a configuration. At any point, if $(u, v) \in E_f$ (the residual network—that is, if $c_f(u, v) > 0$), then we will require $h(u) \leq h(v) + 1$. Any (f, h) pair meeting this restriction (with f a valid preflow and h satisfying $h(t) = 0, h(s) = |V|$) is a configuration.

We have the following **lemma**: in any configuration (f, h), t is not reachable from s in the residual network. If it were, then there would be a simple path from s to t in the residual network, with $\leq |V| - 1$ edges. The node on the path next to s has to have height $\geq |V| - 1$; the next has to have height $\geq |V| - 2$; carrying that forward to t, we get that $h(t) \geq 1$, which is a contradiction.

We also have the following **lemma**: if we have a config (f, h) where no vertices are overflowing, then f is a max flow. The proof is, basically, that if we have no overflowing vertices, then we have that f is a flow; by the previous lemma, t not being reachable implies that f is a max-flow.

We might think that we can just initialize to have 0 flow everywhere, but that violates the requirement that if $(u, v) \in E_f$, then $h(u) \leq h(v) + 1$ (THINK ABOUT THIS A BIT MORE).

We initialize the flow f as follows, then: flow everywhere is 0, except for the edges leaving s; set $f(s, v) = c(s, v)$ for all v. Note we couldn't have done this in Ford-Fulkerson, as this would violate flow conservation; we have a preflow, rather than a flow.

We have a **push** operation, which is valid on an edge (u, v) if (1) u is overflowing; (2) $(u, v) \in G_f$; (3) $h(u) = h(v) + 1$. The natural thing to do, here is to increase the amount of flow on edge (u, v) by $c_f(u, v)$ (or, equivalently, to decrease the amount of flow on (v, u), whichever is possible). We thus change the flow function; this will also change $e(u)$ and $e(v)$. We may have a **saturating** push—which increases the flow by $c_f(u, v)$; we can also have **nonsaturating** pushes, which will occur if $e(u) < c_f(u, v)$: in this case, we can't increase by more than $e(u)$, which will make $e(u) < 0$, violating our nonnegativity condition on preflows.

We need to argue that if (f, h) is a configuration, then (f', h) is also a configuration, with f' the flow after a single push operation. This is pretty clear by the definition of push—we never turn $e(u) < 0$, and we increase $e(v)$.

What if u is overflowing, but we cannot perform a push on any edge (u, v)? Supposing u is overflowing, it must receive at least one unit of flow; it has some neighbor v in the residual network; by backing off that unit of flow, we increase the flow from u to v, so $(u, v) \in G_f$. Thus, there is a nonempty set $v_1, \ldots, v_k$ such that $c_f(u, v_i) > 0$. WE have that $h(u) \leq \min\{h(v_1), \ldots, h(v_k)\}$; if it were otherwise, then we would be able to push to some v (recall teh conditions under which we may perform a push operation).

We thus define a **relabel** operation, which sets

$$h(u) \leftarrow 1 + \min_i \{h(v_i)\}$$

with i ranging over all v_i with $c_f(u, v_i) > 0$. After we relabel, we are guaranteed to be able to push. It's not too tough (I don't think) to see this results in a preflow. Actually, maybe we need another result.

We now upper-bound the number of different number of operations that can be performed, and thereby convince ourselves that we don't get into an infinite loop.

We use the following **lemma**: if u is overflowing, then there is some path from u to s in G_f, for any preflow f. Intuitively, all the flow we "create" happens at the very beginning: if a node has excess flow, it ultimately got it from the source—there should be residual capacity on the reverse path allowing us to push flow back to the source.

Let U be the set of vertices reachable from u in G_f, with u the overflowing node in the network. Suppose, for contradiction, $s \notin U$. Let $\overline{U} = V \setminus U$. We have

$$\sum_{v \in U} e(v) = \sum_{v \in U} \sum_{v' \in \overline{U}} f(v', u) - f(v, v').$$

We don't prove this here, but it's pretty easy to see. Intuitively, if the vertices in U have total excess 100, then it's not from "internal" flows: if we have (u, v) with $u, v \in U$, then a flow of 5 from u to v increases $e(u)$ by 5, but decreases $e(v)$ by the same amount. So all the excess in U comes from $\overline{U}$.

Now, the LHS of the above

$$\sum_{v \in U} e(v)$$

is positive: all the terms are nonnegative, and at least one of them is positive (since $s \notin U$, I believe). Therefore, at least one of the terms on the RHS is positive. Pick such a term $v \in U, v' \in \overline{U}$ with $c_f(v, v') > 0$; that is, edge $(v, v') \in G_f$. Since $v \in U$ (it's reachable from u in the residitual network); further, there is an edge from v to v', then v' is in the residual network. Thus, $s \in U$, and the lemma is proved.

We now have the following **lemma**: No vertex height can exceed $2|V| - 1$. We know the source has height $|V|$. Suppose some other vertex u has height $h \geq 2|V|$. In order for this to have happened, there must have been some relabel operation bumping its height to this value; since the vertex u was overflowing at that time (a condition for relabel), then there was some path from u back to s in the residual graph. There are at most $|V| - 1$ edges on that height; the node next to u on this path has height $\geq h - 1$, the next height $\geq h - 2$, and so on back to the source, with height $|V| + 1$, which is a contradiction. This completes the proof.

From this past lemma, we get another **lemma**: The number of relabels is $O(|V|^2)$, which is pretty easy to see.

Further, we have the following **lemma**: The number of saturating pushess is $O(|V||E|)$. How many times can we do a saturating push from u to v? Suppose we do one–at that point $h(u) = h(v) + 1$, and $c_f(u, v)$ is not in the resulting residual network. The only way that $c_f(u, v)$ can reappear is if we do a push from v to u; in order to do that, the height of v will have increased to be one more than u. We won't be able to do another push from u to v until we've relabeled u again. Thus, for each edge, we can only do $O(|V|)$ pushes ($|V|/2$ maybe, I think).

Note the above argument doesn't hold for nonsaturating pushes: you can do a bunch of nonsaturating pushes without the height of u increasing.

Finally, it remains to bound the number of nonsaturating pushes; we'll want to bound that by $O(|V|^2|E|)$. To do so, we use a potential function argument. Define

$$\sum_{u:e(u)>0} h(u)$$

which is the height of all overflowing vertices (the sink isn't an overflowing vertex, but since the height of the sink is 0, that doesn't matter).

First, note each relabel increases Φ by $O(|V|)$. A relabel doesn't change the flow/excesses—in particular, it doesn't change which nonzero items are summed over in Φ; it only changes the height; since each relabel can only be $O(|V|)$ (since the height is bounded), this gives us the result. Further, this tells us that the total increase due to relabels is $O(|V|^3)$ (there are $O(|V|^2)$ relabels).

Second, each saturated push incrases Φ by $O(|V|)$. In a saturating push from u to v, it may be that v didn't previously have positive excess but does now (that is, the sum can now range over another node; since $h(v) \leq 2|V|$, then the increase is $O(|V|)$). Thus, the total increase due to saturated pushes is $O(|V|^2|E|)$.

Note that initially, $\Phi = 0$; at each point, $\Phi \geq 0$.

Finally, we claim that each nonsaturated push causes Φ to decrease by at least 1. Suppose we have a nonsaturated push from u to v, with $h(u) = k$ and $h(v) = k - 1$. The excess of u, which was positive before, becomes 0; thus the term from u is dropped from Φ, and Φ is thereby made k smaller. We may have, by the nonsaturated push, casued v to contribute to the Φ sum where before it had $e(v) = 0$; thus, $k - 1$ may be added to Φ. If so, the drop is 1, otherwise, the drop is k: either way, we get a positive drop.

13.5 Relabel-to-Front algorithm for max-flow

This is a particular implementation of push-relabel, pushing the pushes/relabels in a better, particular way—this will allow us to make stronger guarantees than we had before.

Recall, above, we had $O(|V|^2|E|)$ as a bound on the number of nonsaturating pushes (which is bad if $|E| \approx |V|^2$; here we'll get an $O(|V|^3)$ bound.

We'll be working with a flow network G, a preflow f in G (recall this means that the net flow into any non-source node ≥ 0); we'll have a height function h such that $h(s) = |V|$, $h(t) = 0$, and, crucially, the property that if $c_f(u, v) > 0$, then $h(u) \leq h(v) + 1$.

We call a valid (h, f) a "configuration".

Define an **admissible edge** (u, v) to be one with $c_f(u, v) > 0$ (there is residual capacity), and $h(u) = h(v) + 1$. Note that if u is overflowing and (u, v)

is admissible, then push(u, v) can be applied.

An **admissible network** is a digraph with vertex set V and the edge set equal to the set of admissible edges. (Note it's not necessarily a subgraph of G, since we could have edges (u, v) in the admissible network not present in G—they may correspond to edges (v, u) along which we can remove flow.)

We have the **lemma** that the admissible network is a DAG—this is pretty straightforward to see, since each edge (u, v) must see one step decrease in h; if we follow a cycle back to the beginning, we have a contradiction.

We have the **lemma** that if (u, v) is admissible and u is overflowing, then PUSH(u, v) is applicable and the operation:

1. Does not add any edge to the admissible network.
2. Might remove (u, v) from the admissible network (because $c_f(u, v)$ is now 0).

Point (1)—that push doesn't create any new admissible edge—is true because the only way that a Push might affect an edge (u, v)'s admissibility in that (u, v)'s residual capacity gets decreased, or (v, u)'s gets decreased. So we might remove (u, v) because $c_f(u, v)$ is now zero (that is, it's a saturating push—this gives us (2) above). Edge (v, u), on the other hand, can't become admissible as the result of a push. This is so because we require that $h(v) = h(u) + 1$ for (v, u) to be admissible; however, if we push from u to v, then we must have that $h(u) = h(v) + 1$. Thus, no push can add edges to the admissible network.

We have the **lemma** that if u is overflowing and there are no admissible edges leaving u, then a RELABEL operation is applicable for u. Further, performing this operation

1. Results in ≥ 1 admissible edge leaving u
2. Results in 0 admissible edges entering u.

Why? Suppose, for example, $h(u) = 10$, and u is overflowing (so there's some positive flow coming in. In other words, we have residual capacity to some neighbors $v_1, \ldots, v_k$, with $c_f(u, v_i) > 0$. We know that $h(v_i) \geq 9$ for all the v_i's; otherwise our height contraint would be violated. If there are no admissible edges leaving u, however, then we know that $h(v_i) =\geq 10$. So we'll relabel $h(u)$ to 1 plus the max of the $h(v_i)$'s, after which there will be an admissible edge leaving u. We know that the height of u will strictly increase; by the height constraint, we get condition (2) above through some argument I don't entirely understand.

We have that each vertex u has:

1. $u.N$, the neighbor list (that is, all v such that $(u, v) \in E$ or $(v, u) \in E$). In other words, this is all v such that $c_f(u, v)$ might be positive, for some f.
2. u.CUR; a pointer to $u.N$, where u.CUR being NIL means "end of list".

We have a function DISCHARGE(u), defined as follows
DISCHARGE(u)

```
• WHILE $u$ is overflowing
    – $v = u$.CUR
    – IF ($v == NIL$):
        ∗ RELABEL($u$)
        ∗ $u$.CUR = $u.N$.HEAD
    – ELSE IF $c_f(u, v) > 0$ and $h(u) = h(v) + 1$ (that is, $(u, v)$ is admissible):
        ∗ PUSH($u, v$)
    – ELSE
        ∗ $u$.CUR = $v$.NEXT-NEIGHBOR
```

Note that v.NEXT-NEIGHBOR refers to the successor of v in $u.N$.

We need go argue that if relabel(u) is called, then there are no admissible edges outgoing from u. Suppose that when DISCHARGE(u) is called, then u.CUR is pointing to the HEAD of $u.N$. I didn't follow this argument at all, augh, check CLRS.

Note we didn't spell out the top-level code, but we will say that, initially, we set u.CUR to $u.N$.HEAD.

So the rest of this section is incomplete. Consult CLRS on the matter.

14 The Hopcroft-Karp algorithm for bipartite maximum cardinality matching

Suppose we have a bipartite graph with sets U and V, with edges E between U and V. A **matching** is a set of edges from the graph such that the edges go from U to V (or vice versa), and no single vertex is incident on more than one edge. The **cardinality** of a matching is the number of edges in it.

Note we will use m to be the number of edges in a bipartite graph (aka $|E|$) and n is the number of vertices ($|U| + |V|$).

We can use, e.g., Ford-Fulkerson to get an $O(mn)$ bound for maximum cardinality matching. (We'll give an $O(\sqrt{n}m)$ solution.

The simple algorithm is as follows. For each edge in E, we create an edge with capacity equal to 1. We also create a source s with edges of capacity 1 pointing to all edges in U; we also make a sink t pointed to by all edges in V. It's not tough to show that max-cardinality matching is equal to finding max flow in the new network. The flow you'd get from Ford-Fulkerson consist of length-3 paths that are vertex-distinct, except for at s and t (think about this).

Ford-Fulkerson is in general pseudopolynomial; however, here, the capacities are all of unit size; since Ford-Fulkerson was $O(f^*)$, and f^* is clearly $O(n)$ here (consider the cut $(s, U \cup V)$, then our algorithm will be $O(mn)$.

What we'll do, then, is reduce the number of iterations to $O(\sqrt{n})$ by finding multiple augmenting paths each iteration (instead of just one). As we did in Edmonds-Karp, we'll only seek the shortest augmenting paths each iteration.

We'll generally take a matching-based view of the algorithm, rather than a flow-based view; however, both make sense.

We iteratively improve our current matching (the empty matching initially) by "flipping" edges (matching to unmatched; vice versa) along an alternating path (which alternates matched/unmatched edge) from an unmatched left vertex to a matched right matching.

An augmenting path alternates between U and V, starting in U and ending in V, and it alternates between matched/unmatched vertices. Suppose a path starts and ends with an unmatched vertex. Supposing it starts in U and ends in V, then flipping this path will increase the cardinality of the matching by exactly 1.

(This has an analogous interpretation in the flow-based view too: an augmenting path is one that goes back and forth between U and V between edges with (u, v) having flow 1 and edges with (v, u) having flow 1, I think).

So suppose our current matching M is not a max-cardinality matching. How do we know that we can find an augmenting path for the matching M? Suppose $|M| = k$, and let M^* be a matching with $|M^*| = \ell > k$. Consider $M \oplus M^*$, with $\oplus$ the XOR operations: the XOR of the two sets is the set of edges that occur in exactly one of M and M^*. The bipartite graph $(U, V, M \oplus M^*)$ consists of vertex-disjoint alternating paths and cycles: a path in this graph will alternate between paths from M and paths from M^*: it could be that a vertex has two edges incident on it, but one of them will come from M, and the other from M^* (why?).

The cycles are of even length, and have equal contributions from M and M^*. Also, there must be at least $\ell - k$ paths corresponding to vertex-disjoint augmenting paths in M; each of them will be of odd length (I think? WHY?). That is, augmenting paths must exist if M isn't an MCM (max-cardinality matching). How do we efficiently find these augmenting paths, though?

The Hopcroft-Karp algorithm is as follows.

- $M = \emptyset$
- while there is an augmenting path WRT M:
  - Find a maximal set of shortest vertex-disjoint augmenting paths WRT M (call these $P_1, \ldots, P_k$).
  - Augment M: $M = M \oplus (\bigcup_i P_i)$

Note that the "maximal set" referred to above is "maximal" in the sense that there are no other vertex-disjoint augmenting paths; a bigger one may exist—maximal here doesn't mean "largest cardinality".

We have the following **lemma**: Let ℓ_i be the length of the shortest augmenting path in iteration i; then $\forall i . \ell_{i+1} > \ell_i$; that is, the shortest-augmenting-path-lengths are strictly increasing. (proved later)

After $\sqrt{n}$ iterations, the shortest augmenting path will be of length $\sqrt{n}$; once we reach this path, we can argue that even if we were to just use Ford-Fulkerson until we get an MCM, we'll be done within $\sqrt{n}$ iterations. If we're k away from a MCM; then there exist k vertex-disjoint augmenting paths, each of which is of length at most $\sqrt{n}$, so k is at most $\sqrt{n}$ (what?). In other words, our current matching size is within $\sqrt{n}$ of an MCM.

So the two things we need to do are (1) prove the above lemma; (2) show how we can find a maximal set of shortest vertex-disjoint augmenting paths in $O(m+n)$ time.

Let's think about (2). Finding a single shortest augmenting path isn't too tough. Supposing we have some unmatched vertices in U (and some in V); we start with the unmatched ones in U, and look at all their neighbors in the graph. If one of them were connected directly to an unmatched vertex in V, then we would have found an augmenting path of length 1; otherwise, we have an edge from an unmatched U vertex to a matched one in V; by definition, we'll follow the outgoing edge from this node into the matched set in U; we'll next examine all the neighbors of that node. So what we have is a BFS; however, when we reach a node in V, instead of following all outgoing edges, we only follow the matched edges into U (recall we're looking for an alternating path, not just any path).

Generalizing this to finding a maximal set of shortest augmenting paths is tricker. Suppose we've found a single shortest augmenting path. We then know the length. What we can do, then, is look at the unmatched vertexes in V; suppose our shortest-matching path length is 3; we'll look for edges back to the matched section in U such that (1) the destination vertex in u has label 1 minus 3, and (2) the path is vertex-disjoint from all the previously-found shortest-augmenting-paths. Thinking about this a bit shows it isn't too hard—we basically keep some extra data around in our initial BFS, and use it when we work backwards.

Back to (1), proving the above lemma. Suppose $P_1, \dots, P_k$ is a maximal set of vertex-disjoint shortest augmenting paths wrt M. Let M' denote $M \oplus \cup_i P_i$, the matching after we've augmented it in an iteration. Let P be a shortest augmenting path wrt M'. (P is a candidate for our next iteration; we need to show that the length of P is greater than ℓ.)

Let $A = M \oplus M' \oplus P$, and let $M^* = M' \oplus P$. So, in other words, since $\oplus$ is associative, we have that $A = M \oplus M^*$. Let $|M| = s$; we have that $|M'| = S + k$, and $|M^*| = s + k + 1$.

Recall that if we have a matching of cardinality k, and one of cardinality $\ell > k$, then XORing them gives paths and cycles, with at least $\ell - k$ augmenting paths wrt the mathcing. In particular, with $A = M \oplus M^8$, we have that it contains at least $(+k+1) - s = k+1$ vertex-disjoing augmenting paths wrt M. Each path has length $\geq \ell$, so we have that $|A| \geq \ell(k+1)$.

We associate A the other direction: since $M \oplus M' = \cup_i P_i$, we have $A = (\cup_i P_i) \oplus P$. Since each P_i has exactly ℓ edges, and they're vertex-disjoint (and therefore edge-disjoint), we know that

$$|\cup_i P_i| = k\ell.$$

This, together with the fact that $|A| \geq \ell(k+1)$, imply that $|P| \geq \ell$. To show our result, we need to rule out the possibility that $|P| = \ell$. Assume $|P| = \ell$. There are two cases:

- P is vertex-disjoint from all the P_i's. Then the P_i's are not a maximal set: you could put P in there and have a larger set.

- There is some i such that P shares a vertex with P_i. Let u be the necessary vertex that belongs to both P and P_i. Since $u \in P_i$, then u is matched in M'. This is so because, after augmenting along a path P_i, then every single vertex on the path P_i is matched; thus, u is certainly matched. Let u have associated matched edge $(u, v) \in M'$. P is an augmenting math wrt M'; since $u \in P$, this implies that the edge (u, v) is on P (no augmenting path stops at a matched node and then doesn't follow the matched edge). Therefore, (u, v) was an unmatched edge in M, and is on P_i (So P_i flipped the edge (u, v)). We have identified an edge on both P and P_i. Thus, the common edge (u, v) will be dropped out of $A \equiv \cup_i P_i \oplus P$.

  In particular, if $|P| = \ell$ but has a nonempty intersection; then the size of A is less than $k\ell + \ell$ (since the cardinality of the $\cup_i P_i$ set is exactly $k\ell$; this contradicts the already proven statement that $|A| \geq \ell(k+1)$, and we have a contradiction.

Consult CLRS on how to find a maximal set of augmenting paths (I think it's in there?). Intuitively, at each node, we keep track of how many predecessors that node has in the BFS tree; when we remove a path, then, we decrease the predecessor counts appropriately; this prevents us from running into a dead-end on our path back to the start.

15 Scaling algorithms for SSSP

Recall m is the number of edges, n is the number of vertices in a graph. Letting U be the max edge weight, we can ger running time

$$O(m \log_{2+m/n} U).$$

Gabow (1985) give a result like this; he has a 1989 paper with Tarjan that improves on this. Note that if we can bound U by, e.g., n^{10}, then the logarithmic term is just a constant.

A naive implementation of Dijkstra's algorithm is $O(m \log n)$ (which is $O(|E| \log |V|)$). This is the solution where we use a simple heap to implement the priority queue. Using a faster pqueue (e.g. a Fibonacci heap or a relaxed heap), we get Dijkstra's algorithm down to $O(|E| + |V| \log |V|) = O(m + n \log n)$.

15.1 Algorithm 1

Let d_i be the desired shortest path distance from s to i.

This algorithm is based on the notion of a z-near optimum solution, which is a set of nonnegative integers e_i such that

- $e_s = 0$.
- $\forall (i,j) \in E$, we have $e_i + \ell_{ij} \geq e_j$, with ℓ_{ij} the length of (i,j).
- Each vertex i has a path from s of length between e_i and $e_i + z$.

Call these conditions (1), (2), and (3).

We claim that for all i, we have $e_i \leq d_i$. Let P be a shortest path to i. We have the inequality that $e_u \leq d_u$ (somehow); further, condition (2) gives us that $e_u + \ell_{ui} \geq e_i$ (somehow).

So for P a shortest path, we have

$$\begin{aligned} d_i &= d_u + \ell_{ui} && (30) \\ &\geq e_u + \ell_{ui} && (31) \\ &\geq e_i && (32) \end{aligned}$$

I was wrestling with fucking TexShop, so missed this proof.

Now, we construct a modified graph G' with edge weights

$$\ell'_{ij} = e_i + \ell_{ij} - e_j \geq 0$$

where we get nonnegativity by (2). Similar to what we had in Johnson's APSP algorithm, the modified edge lengths preserve shortest paths; that is, paths from s to i are shifted by $-e_i$ (it's a similar concept to how it was before—add the weights for the path weights, and you get a telescoping sum).

So we can calculate shortest paths in G', which has the nonnegative edge weight property; we can simply shift back:

$$d_i = d'_i + e_i$$

to get the shortest paths in the original graph G.

We have the nice property that, with d'_i the d_i's in G', that

$$0 \leq d'_i \leq z.$$

If z is "small" (for example, we will have $z = O(m)$), then we can implement the priority queue of Dijkstra's algorithm in a trivial manner. In particular, we will us an array of linked lists, where the entry at index t is a linked list giving all vertices with distance label t; we only have to maintain these up to z. Running Dijkstra's algorithm, then, we scan this from left-to-right: rather than doing a delete from a pqueue (which will be $O(n)$), we'll just need to look at the bucket where we're currently working.

Normally, we wouldn't be able to do this trick, because we could in general have to skip past a whole bunch of empty buckets; however, now since we have $O(m)$ buckets, the total cost of the delete-min operations will be linear.

First, we'll show an $O(m \log_2 U)$ algorithm; we'll then show the result claimed at the beginning of this section.

First, look at all edge weights; if all of them are $\leq m/n$, then set each $e_i = 0$. This is m-near-optimum: looking at conditions (1)–(3), we see they are all satisfied. Otherwise, we will modify G to obtain $\overline{G}$, where

$$\overline{\ell}_{ij} = \lfloor \ell_{ij}/2 \rfloor$$

where $\overline{G}$ has the same structure as G. Calculate the $\overline{d}_i$'s recursively (if the edge weights are small, do it the fast weight, otherwise perform the $\overline{G}$ operation again).

Now, let $e_i = 2\overline{d}_i$. We claim the the e_i's are m-near-optimum:

1. (1): $e_s = 0$, trivially.
2. (2): We have $\overline{d}_i + \overline{\ell}_{ij} \geq \overline{d}_j$, since the path distances satisfy condition (2). Multiplying the above by 2 gives us $2\overline{d}_i + 2\overline{\ell}_{ij} \geq 2\overline{d}_j$. Since we have $\ell_{ij} \geq 2\overline{\ell}_{ij}$, we get $2\overline{d}_i + \ell_{ij} \geq 2\overline{d}_j + 2\overline{d}_{ij} \geq 2\overline{d}_j = e_j$.
3. (3) we must show $d_i \leq e_i + n$ (we'll actually show it's n-near-optimal); note that $e_i = 2\overline{d}_i$.

   Consider a shortest path from s to i in $\overline{G}$; by definition, its length is $\overline{d}_i$. Consider an edge (u, v) on this path. We have $\ell'_{uv} = \lfloor \ell_{uv}/2 \rfloor$. We therefore have

   $$2\ell'_{uv} \leq \ell_{uv} \leq 2\ell'_{uv} + 1$$

   That is, ℓ_{uv} is either $2\ell'_{uv}$ or $2\ell'_{uv} + 1$. So the path P in G is the sum of the ℓ_{uv}'s; since each ℓ'_{uv} is that or that + 1, then we have P with length $\leq 2\overline{d}_i + n - 1$ (what? why? what?)

   We therefore have that $d_i \leq 2\overline{d}_i + n - 1$.

At each level of the recurrence, we'll be spending $O(m)$ time to obtain the d_i's from the $\overline{d}_i$'s; there are $\log_2 U$ levels, since after that many iterations, we'll be dealing with 1-bit numbers. Overall, then, we have $O(m \log_2 U)$ time.

We can improve this to $O(m \log_{2+m/n} U)$ time. Intuitively, instead of dividing the ℓ_{ij}'s by 2 at each stage, we'll instead divide by $2 + m/n$; if the graph is somewhat dense, then this is a significantly bigger reduction at each step.

If we divide by $x = 2 + m/n$, (that is $\overline{\ell}_{ij} = \lfloor \ell_{ij}/x \rfloor$), then we'll be able to claim that $e_i = x \cdot \overline{d}_i$ is m-near optimal (rather than n-near-opt).

At one step, we incur an error of at most x (that is

$$\ell_{uv} \leq (x+1)\overline{\ell}_{uv})$$

Thus, since we have n vertices, then the "z" that we have z-near-optimum for is approximately $nx = \Theta(m)$.

15.2 Algorithm 2

The last algorithm had the property that we were working with potentially very large numbers: $\log U$ could be potentially much larger than $\log n$.

We have k-bit ℓ_i's, where k is approximately $\log U$. Write ℓ_i out as its bits $b_1, \ldots, b_k$, with b_1 the highest-order bit.

- We first view the weights as their MSB's, the b_i's. Call that graph G_1. We then compute the d_i''s in G_1. That is,

$$\ell_{ij}^{(1)} = b_1$$

  where b_1 stands for the MSB from the appropriate weight.

- We next scale as follows:

$$\ell_{ij}^{(2)} = 2(d_i' + \ell_{ij}^{(1)} - d_j') + b_2$$

  This is better than the obvious thing to do, which is

$$\ell_{ij}^{(2)} = 2\ell_{ij}^{(1)} + b_2.$$

  Using the first scheme, we get the numbers confined to a much smaller range; in the latter, more obvious scheme, we get possibly very large numbers.

The key claim is that

$$\forall t. 1 \leq t \leq k \left[d_i^k = O(n) \right]$$

So at each stage, we can run Dijkstra's algorihm where all the weights are $O(n)$. Or something.

We can show without too much difficulty that

$$d_i = \sum_{t=1}^{k} 2^{k-t} d_i^t$$

That is, once we've calculated d_i^t for all the different stages t, we can somewhat easily arrive at the numbers we're actually interested in computing.

16 Linear Programming

We're seeking to optimize a linear function subject to linear constraints. WLOG, we can assume the following form:

- Maximize $c^T x$
- subject to $Ax \leq b$ and $x \geq 0$.

We assume c and x are $n \times 1$, b, y are $m \times 1$, and A, which is given, is $m \times n$.

For example, suppose we're looking to maximize $3x_1 + 2x_2$, subject to the three constraints

$$\begin{aligned} x_1 + x_2 &\leq 10 \\ x_1 &\leq 8 \\ x_2 &\leq 5 \\ x_1, x_2 &\geq 0 \end{aligned}$$

There is a simple geometric view here: draw the various lines giving the constraints, and shade the appropriate halfplane—that gives us our feasible region. We then consider the family of lines fo the form $3x_1 + 2x_2 = c$, and we ask what the highest value of c is of such a line intersecting the feasible region. This value will be 28.

There is, of course, the simplex method. Intuitively, it works as follows: we know that an optimal solution will occur at a vertex of the simplex. We greedily move from one vertex to the next, looking for a solution. There are pathological cases where there are a ton of vertices, and we slowly spiral around to the optimal solution.

It's not in general polynomial, but works quite well in practice. If we look at "smoothed complexity", however (not defined here), the simplex method has polynomial smoothed complexity. Intuitively, supposing we take an LP instance, and we perturb the numbers very slightly, then we "break" the pathological cases that are problematic for the simplex method. In other words, these pathological cases are not robust to noise—perturbing them slightly makes simplex work fine.

Note there are methods dating back to the 70s for solving arbitrary LP instances in polynomial time (e.g. ellipsoid algorithm, interior point methods). Linear programming is a nice, powerful tool for showing problems to be polynomial-time solvable (e.g. multicommodity flow, where we have various commodities flowing in our network, rather than just a single commodity).

Note we could have put equality constraints in our formulation of LPs above, as we can model equalities as two inequalities. We could also have given the formulation with equalities, and modeled inequalities with slack variables. That is, we have a number of forms that we can choose to formulation the notion of LPs.

In general an LP can fall into one of three types:

1. infeasible (when the feasible region is empty)
2. feasible (bounded)—the function is bounded
3. feasible (unbounded). Consider, e.g., $\max x_1$ s.t. $x_1 \geq 0$.

Note the **dual** LP problem is as follows. Suppose the primal is $\max c^T x$ s.t. $Ax \leq b, x \geq 0$. Then our primal has n constraints.

The dual is $\min y^T b$ s.t. $A^T y \geq c, y \geq 0$. It's easy to show that the dual of the dual is the primal.

We have the a result called **Weak Duality**: Suppose x is feasible for the primal and y is feasible for the dual. Then

$$c^T x \leq y^T b.$$

The proof of this as follows. Since we have, from a dual constraint, that $A^T y \geq c$, then we also have $y^T A \geq c^T$. Thus, from the primal, we have

$$\begin{aligned} c^T x &\geq (Y^T A)x && (33) \\ &= y^T (AX) && (34) \\ &\leq y^T b. && (35) \end{aligned}$$

We point out two intresting things weak primality entails. This implies, first, that if the primal is unbounded, then the dual is infeasible. Second, we have that if x is feasible for the primal, y is feasible for the dual, and $c^T x = y^T b$, then we have that x is optimal for the primal and y is optimal for the dual.

16.1 Example

We continue the initial example given in the previous subsection.

The dual formluation for the primal LP given in the previous section is:

$$\min 10y_1 + 8y_2 + 5y_3$$

s.t.

$$\begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} \geq \begin{pmatrix} 3 \\ 2 \end{pmatrix} \quad (36)$$

$$y_1 \geq 0, y_2 \geq 0 \quad (37)$$

We view the matrix above as two constraints ($y_1 + y_2 \geq 3$, etc). In an optimal solution, we'll have the two inequality constraints being tight. That is, we'll have

$$\begin{aligned} y_1 + y_2 &= 3 \\ y_1 + y_3 &= 2 \end{aligned}$$

From these, we deduce

$$y_2 = y_3 + 1$$

So what we have (why?) is

$$\min 10y_1 + 13y_3 + 8$$

s.t. $y_1 + y_3 = 2$ and $y_1, y_3 \geq 0$. We'll have $(y_1, y_2) = (2, 0)$, and our objective function is $20 + 0 + 8 = 28$.

Note that we got the same answer for the dual as we got for the primal. This is an instance of strong duality.

16.2 Strong Duality

We have the following result, called **strong duality**. If an LP is feasible and bounded, it has the same optimal objective function as its dual (which is also feasible and bounded).

16.3 Max-flow as LP

We'll end up with the fun result that the max-flow min-cut theorem may be viewed as a special case of strong duality: the dual of a max-flow problem will end up being equivalent to solving min-cut.

Our variables are as follows. For every edge $e \in E$, we have a variable x_e (so $x \in \mathbb{R}^{|E|}$), and x_e is the amount of flow on edge e. We have $\forall e x_e \geq 0$, by the definition of flow. Further, we have the constraint $\forall e$, $x_e \leq$ the capacity of edge e. We also have $|V| - 2$ constraints telling us that

$$\forall u \in V \setminus \{s,t\}. \sum_v x(u,v) = \sum_v x(v,u). \tag{38}$$

We can write the above equality as two inequalities (further, we can get a constant on the RHS of the equation, as needed for our definition of LP, with simple algebra). The objective function is

$$\max \sum_v x(s,v) - x(v,s)$$

that is, we maximize the sum of the net flow out of the source.

The dual formulation is as follows. We have variables α_e for each capacity constraint. We have variables β_u, γ_u for each of our constraints of type (38) (we need two variables because we're giving our equality cosntraints as two inequalities).

We then have the following dual objective

$$\min \sum_e \alpha_e$$

that is, we minimize the capacity of each edge e. The dual constraints are

$$\forall u,v \in V \setminus \{s,t\}. \alpha(u,v) + (\beta_u - \gamma_u) + (\beta_v - \gamma_v) \geq 0 \tag{39}$$

$$\alpha(s,t) \geq 1 \tag{40}$$

$$\alpha(t,s) \geq -1 \tag{41}$$

$$\forall e. \alpha_e \geq 0 \tag{42}$$

$$\forall v \in V \setminus \{t\} \alpha(u,v) - (\beta_v - \gamma_v) \geq 1 \tag{43}$$

We also have a bunch more constraints that look like (43) with t replaced with s, and whatnot.

It could be clearer to write $\beta_u - \gamma_u$ as δ_u (since that's how they always appear). Intuitively, since we're modeling equalities, we have the unrestricted variable δ_u, which isn't required to be nonnegative. This is what the more general dual formulation looks like anyways when we have equality constraints in the primal: we'll end up with an unrestricted variable δ_u anyways.

Note we could also model all the ugly special inequalities (43) by unifying the dual constraints as

$$\forall u, v. \alpha(u, v) + \delta_v - \delta_u \geq 0$$

s.t. $\delta_s = 1, \delta_t = 0$.

We'd like to claim that this is the min capacity cut problem. Our problem looks like that—we're summing over capacities α_e. We can reason that in an optimal solution,

$$\forall e \alpha_e = \max(0, \delta_u - \delta_v)$$

We can thus substitute in for the α_e's in the problem formulation, and we just have a problem with variable δ_u, δ_v.

WE'll be able to make the following claim: there is an optimal solution where all the δ_u's are ≤ 1. Further, there's an optimal solution where all the δ_u's are ≥ 0. Finally, we'll be able to claim that there's an optimal solution where all the δ_u's are either 0 or 1.

Once we've done that, we'll be able to interpret this as the min-capacity cut problem. THis is so because choosing δ_u to be 0 or 1 means choosing u to be on the source side of the cut or the sink side of the cut.

17 NP-completeness

17.1 Integer Linear Programming

ILP is just like LP, but the variables are required to take on integer values. It may seem that, since there are "fewer choices" for the variables, this would be easier; however, it ends up the case that no poly-time algorithm is known (the Decision Procedure version of the problem is NP-complete, actually).

So for LP, a local optimum was also a global optimum. However, this isn't necessarily the case in ILP.

We may think that we can just ignore the integrality constraints—solve the general linear program, and round. In a few restricted settings, this actually comes close sometimes. In general, though, this can yield a solution that is either infeasible, or feasible but not optimal.

In fact, even 0-1 ILP is hard (that is, ILP but where we're restricted to the numbers 0 and 1). Intuitively, we wouldn't expect rounding to work at all here.

It's actually somewhat easy to write a knapsack instance as a 0-1 ILP. The knapsack problem, recall is as follows. Item i has value v_i and weight w_i. The Knapsack has a capacity W. To formulate this as a 0-1 ILP, we define variables

x_i, which take value 1 if we take item i, and 0 otherwise. We solve

$$\max \sum_i x_i v_i \tag{44}$$

$$s.t. \sum_i x_i w_i \leq W \tag{45}$$

$$x_i \in \{0, 1\}. \tag{46}$$

So we have a nice problem that, if we slightly tweak it, we get something that's not solvable in polynomial time.

17.2 NP

If we can give a polytime algorithm for any NP-complete algorithm, we get a polytime algorithm for any of the other ones (so if we solve knapsack, for example, we've solved ILP, as we'll show).

As usual we restrict our attention to Decision Problems (DPs)—it's easier to restrict ourselves to questions that all have the same output format (that is, they all simply answer "yes" or "no").

This might feel weaker—suppose we have knapsack, and what we really want is an encoding of an optimal set of items to take. We might think that the decision version of this problem is easier (which, in this case, is supposing we have a knapsack instance plus a bound B, is there a set of items that fits in the backpack and has value at least B). It's not tough to see this isn't the case—if we could solve the DP version in polytime, then we could solve the original version in polytime—do bsearch to figure out the exact optimal value that can fit in the knapsack; to figure out exactly which items fit in the knapsack, delete an item, and see if the optimum goes down—if it does, then we know an item must be in the optimal solution. Thus, by a polynomial number of calls to the decision version, we can solve the optimization version, so the latter is no harder than the former.

We may think of DPs as a binary language over $\{0, 1\}^*$. For example, consider the DP version of knapsack described above—can a set of items with value $\geq B$ fit into the knapsack of capacity W? It's somewhat easy to see how we could encode an instance of knapsack into a binary string—as long as we choose some reasonable encoding, the complexity will be the same, and we can view the problem as taking bitstrings as input.

So we talk below about the class P, which are *decision problems* that are solvable in polynomial time. In other words, there should be a polynomial time algorithm which, polynomial in the length of the input, terminates with yes or no.

The class NP is as follows. A language L is in NP iff it is polynomial-time verifiable. There is an asymmetry baked in here: "yes" instances of the problem have a short proof allowing us to verify that they in fact work; there isn't necessarily such a proof for "no" instances.

More formally, $L \in NP$ iff there is a two-input polynomial-time verifier $A(x, y)$[1] such that the following two things hold:

- $\forall x \in L.\ \exists y.\ |y| = O(|x|^c)$ for some constant $c > 0$ and $A(x, y) = 1$; and
- $\forall x \notin L.\ \forall y.\ A(x, y) = 0$ (that is, there is no proof that can fool us into thinking $x \in L$).

Note that if $L \in P$ then $\overline{L} \in P$. However, because of the asymmetry in the two bullet points above, it isn't necessarily the case that the above holds for NP. That is,

$$L \in NP \Leftrightarrow \overline{L} \in coNP$$

and certainly we don't know if $coNP = NP$.

17.2.1 Example: SAT

As an example, we have that $SAT \in NP$. Supposing we have a boolean formula x using the connectives $\wedge, \vee, \neg$. We have that $SAT \in NP$, pretty easily: all we need to do is exhibit a poly-time verifier algorithm A that satisfies the definition given in the previous section. How does this work? It takes as two strings, x (the SAT instance), and y (the truth assignment—let $|y|$ be the number of variables in x; we interpret y as a truth assignment). The algorithm A need only evaluate the formula with the truth assignment given in y, which can be done in polynomial time.

Note the verifier need not come up with a truth-assignment y; it need only check whether a given assignment is true. That's enough to satisfy the definition to show SAT $\in NP$.

17.2.2 Example, 0-1 ILP, DP version

Let's think now about the decision version of 0-1 integer linear programming. The input is A, b, c, with A an $m \times n$ matrix; b and c $m \times 1$ vectors. To make it a yes-no question, we give ourselves a bound $B \in \mathbb{Z}$ and ask: is there an $x \in \{0, 1\}^n$ s.t. $Ax \leq b$ and $c^T x \geq B$ (that is, is there a feasible solution with objective at least B?).

To show 0-1 ILP is in NP, we'll treat y as a 0-1 assignment to the n variables x_i. We then check if the given y is feasible, and then check to see if the objective is $\geq B$ (both of which can be done in polynomial time pretty straightforwardly).

17.2.3 Example, UNSAT

We may think that any problem that isn't polynomial-time solvable is in NP. Of course this isn't true. Consider the problem UNSAT, which is the complement of SAT— answer ”yes” iff a problem is unsatisfiable. How could we give a short proof that a formula is not satisfiable? We don't show anything here, but it's

[1] x is the string we want to determine if it's in the language; y is the purported proof that $x \in L$. The job of the verifier is to check whether y is a valid proof that $x \in L$.

certainly not clear that UNSAT is in NP (and, in fact, isn't assuming NP $\neq$ co-NP, though we don't know that).

17.3 Polynomial-time transformations

Consider a problem X (for example, SAT). There are many possible instances of X, some of which have answer "yes" and some of which have answer "no" (and similarlty with another problem Y). To give a polynomial-time transformation from X to Y, we must give a mapping from every $x \in X$ to some $y \in Y$ such that, for every "yes" instance x, we map to a "yes" instance in Y (and ditto with the "no" instances). Note that this transformation need not be either injective or surjective: the image of the transformation may be a proper subset of Y.

That is, we require, $\forall x \in \{0,1\}^*$, $x \in X$ iff $f(x) \in Y$.

If we have such an f that is poly-time computable, we write

$$X \leq_P Y.$$

If Y can be solved, for example, in polyonimal time, then we can also solve X in polynomial time: take an instance of X, send it to Y, and solve it. So the above is $\leq_P$, rather than $\geq_P$, because a polynomial-time transformation means that X is at least as easy as Y, in the obvious coarse sense.

For example, if we show that $X \leq_P Y$ and $Y \in P$, then we can conclude that $X \in P$. That is, we solve problem X by reducing it to a problem Y we already know how to solve.

We also have the following related property, which is basically the contraposed version of the above. If $X \leq_P Y$ and $X \notin P$, then $Y \notin P$ (these results are typically more difficult to find). For example, we showed KNAPSACK $\leq_P$ 0-1 ILP. If we were to show that KNAPSACK is not solvable in polynomial time, then we'd also have that 0-1 ILP is non-polytime solvable.

17.4 NP-Completeness

Now, we have that $L \in NPC$ iff $L \in NP$ and $\forall L' \in NP.\ L' \leq_P L$.

This seems hard because of the $\forall$ on the RHS of the biconditional. We get around this with the Cook-Levin theorem, which tells us SAT $\in NP$.

Once we have a particular NP-complete problem, we can reformulate the above statement to

$$L \in NPC \Leftrightarrow L \in NP \wedge \exists L' \in NPC.\ L' \leq_P L.$$

So that is, when we have a new problem, we don't go through, for every language $L' \in NP$, and prove that $L' \leq_P L$; instead, we have a single language that we go through. That is, to show L is NP-complete, we use

$$L \in NP \wedge L' \leq_P L \wedge L' \in NPC \Rightarrow L \in NPC.$$

Be sure to get the order right. Reduce **from** SAT, not to it. If we were to show L's complexity is less than some NP-complete problem, that doesn't imply that it's a hard problem, but merely easier than some hard problem.

17.4.1 CIRCUIT-SAT, showing SAT NP-complete

CLRS proves that, for any $L \in NP$, L is polynomial-time reducible to circuit-SAT. The circuit-SAT problem is as fast: suppose we're given some boolean circuit, with no feedback/cycles. It has $\wedge, \vee, \neg$ gates, for example, and has a single output wire. The problem is as follows: given a circuit, is there some input to the input wires that makes the output wire true.

We'll show that CIRCUIT-SAT $\leq_P$ SAT; in the latter, the input is a boolean formula, not a circuit, so we have to convert from circuits to formulas.

It's easy to show that SAT $\in$ NP: given a truth assignment to the variables, check in polynomial time whether the formula evaluates to true or false in the assignment.

We now show CIRCUIT-SAT $\leq_P$ SAT. We need a polytime computable f mapping x to a formula $f(x)$ such that

$$x \in \text{CIRCUIT-SAT} \Leftrightarrow f(x) \in \text{SAT}.$$

To convert from x to a corresponding boolean formula $f(x)$, there's a pretty obvious thing to do: topologically order the gates in the input circuit, and, at each gate, write an expression for the value of the output wire, in terms of its inputs. Defining $f(x)$ in this way does satisfy the condition above: it's a sound transformation or whatever. However, it doesn't yield a polynomial-time computation: we can come up with fairly simple examples whose output length is exponential in the input length: input x_1, x_2 to an AND gate, split the output of that into another AND gate (doubling the output size), and repeat this: at each AND gate, you double the size of the formula.

WE get around the above by introducing a variable for each gate: define a new variable for the output wire for each gate. Even if there's a huge circuit underneath a gate, the formula at the gate will have a constant-length formula (rather than before, where the subformula at each gate was potentially explosive). To combine them, we simply take the conjunction of all the fragments (so an AND gate taking x_1 and x_2 will have the formula $y_1 \leftrightarrow x_1 \wedge x_2$, for example), and conjoin the fact that the output wire (which has a variable, y_k, say) must take value 1. That is, if gate i has local formula ϕ_i, then our output formula is

$$y_k \wedge \bigwedge_i \phi_i.$$

This formula is satisfiable iff the original circuit had a satisfying input.

In one direction, assume the circuit is satisfiable. Then there is an assignment making the boolean formula above true. To see this, look at what each of the output wires of the gates carry: set the x_i's (the original input variables) equal to the satisfying interpretation we're given; set the y_i's equal to what they are in the original circuit. All of the little fragments (referred to as ϕ_i above) will evaluate to true—they basically just encode the meanings of the gates. Further, the y_k at the top of the formula (the output wire) will be true, since we assume we have a satisfying assignment.

On the other hand, suppose there is a satisfying assignment to the $y_k \wedge \bigwedge_i \phi_i$ formula. Then it is pretty easy to see that the x_i portion of that is a satisfying interpretation of the circuit: the y_j's represent exactly the outputs of the individual gates.

So we have given a polytime computable f that has the properties we want, and have shown SAT to be an NP-complete problem (assuming that CIRCUIT-SAT is NP-complete, which is shown in the text).

17.4.2 3-SAT is NP-Complete

3-SAT is just like SAT, but we require the formulas to be in 3-CNF. A 3-CNF formula is of the form

$$\bigwedge_i C_i$$

where each $C_i = (\ell_{i1} \vee \ell_{i2} \vee \ell_{i3})$, with ℓ_i giving a literal (a variable or the negation of a variable), and each of the literals in a clause involve distinct variables.

Frequently, a problem is shown to be NP-complete via a reduction from 3-SAT, so it's pretty important to see 3-SAT is NP-complete; that is, 3-SAT $\leq_P X$ is often used to show show X is NPC. Why don't we just reduce from SAT instead of 3-SAT? Well, a reduction from 3-SAT is basically always easier than a reduction from SAT: reducing from SAT requires that we handle all possible formulae; reducing from 3-SAT requires considering only a very restricted subset of the formulas. That is, a reduction from 3-SAT won't possibly make the proof any more difficult.

To show 3-SAT NP-complete, we must first show it's in NP: since it's a restricted version of SAT, of course it is in NP.

Now, to show SAT $\leq_P$ 3-SAT, we need a polynomial-time computable f with the desired properties. Let x be a given SAT instance (assume it uses $\wedge, \vee, \neg$). Construct the parse tree for x (built as you think it is). Use the same trick as we used for CIRCUIT-SAT: for each node in the parse tree, introduce a new variable y_i, and introduce conditions for its verification: if we have a node for $x_1 \vee x_2$, for example, we'll have $y_1 \leftrightarrow x_1 \vee x_2$, for exampls. We then set $f(x)$ equal to the conjunction of the output variable (y_k, say) and the various clauses. Again, calling the clause introduced at node i ϕ_i, we have

$$f(x) \equiv y_k \wedge \bigwedge_i \phi_i.$$

For a similar reason to that given in the last construction, $f(x)$ is poly-sized.

Now, $f(x)$ isn't quite a 3-CNF formula yet, but we do have the conjunction of a number of clauses, where each of the clauses involves 3 variables or fewer. We can fairly straightforwardly convert it to 3-CNF: we only have a few types of formulas (one form from $\vee$, one from $\wedge$, one for $\neg$). $\vee$ and $\wedge$ are somewhat simple; the ones introduced by $\neg$, of the form $y_i \leftrightarrow \neg x_j$, are converted to 3-CNF clauses by introducing dummy-variables. To get the idea, consider the root variable y_k; to rewrite that as the conjunction of 3-CNF clauses, write it as

$$(y_k \vee a \vee b) \wedge (y_k \vee a \vee \neg b) \wedge (y_k \vee \neg a \vee b) \wedge (y_k \vee \neg a \vee \neg b)$$

which, it is easy to see, is logically-equivalent to y_k. The negations with only 3 variables use a single dummy similarly.

17.4.3 Maximum Independent Set (MIS) is NP-Complete

Given an undirected graph G, and an integer k, we ask: does G have an independent set of size $\geq k$. An Independent set is a set of vertices such that no two are adjacent to each other. (This is obviously the decision-problem version of the typical MIS formulation).

It's pretty easy to see that MIS $\in$ NP. Suppose we have a positive instance: we have a graph G that does have an independent set of some k. A short proof of that fact is basically just an encoding of the independent set: a bit-vector of cardinality $|V|$, for example.

Now, we show 3-SAT $\leq_P$ MIS; that is, if MIS were polynomial-time solvable, then we'd have a poly-time solver for 3-SAT. Suppose we have a 3-SAT instance x. We need to map it in polynomial time to an MIS instance $f(x)$ such that $x \in$ SAT iff $f(x) \in$ MIS.

The input is a 3-CNF formula; we must transform that into a graph G and integer k. Set k to the number of clauses in x; to build the graph G, we start with a triangle for each clause: each clause C_i has a triangle, and each literal in the 3-clause has a vertex. We must add some additional edges: we put edges between any nodes u, v where the label of u is the negation of v. It's pretty easy to see that, given an arbitrary 3-CNF formula, this construction takes polynomial time.

We must show both directions. First, suppose $x \in$ 3-SAT; that is, we can come up with a truth assignment to the variables such that at least one of the literals in every vertex is satisfied. Pick one satisfied literal arbitrarily out of each of the clause-triangles. The circled vertices must form an independent set of size at least k. It's clearly at least of size k. They're also independent; if there were an edge from two vertices not in the same triangle, then one is the negation of the other, and it wouldn't be a satisfing assignment.

On the other hand, suppose $f(x) \in$ MIS. Then we have an independent set whose size is at least the number of clauses. Then we choose at least one vertex out of each triangle. We have k triangles, and we can't take two or more vertices from each vertex (it's an independent set). From that, we can read off a satisfying assignment to x: for each node in the independent set, look at its label: if a node is labeled x_i, then set x_i to true; if it's labeled $\neg x_i$, set x_i to false. By doing so, we have that each clause is satisfied. It can never be the case that we encounter both x_i and $\neg x_i$; they have an edge between then, and can never occur in an independent set.

It is worth noting that this proof would have been much uglier if we had to consider SAT in general.

17.4.4 Vertex-Cover (VC) is NP-Complete

This is almost exactly the MIS problem. A vertex cover for a graph $G = (V, E)$ is a subset $U \subseteq V$ such that

$$\forall (u, v) \in E.\ \{u, v\} \cap U \neq \emptyset.$$

That is, each edge in the graph has at least one of its endpoints in the set U. Clearly, V is a vertex cover, but we can often get smaller vertex covers.

There's a close relationship between VC and MIS: U is an independent set of $G = (V, E)$ iff $V \setminus U$ is a vertex cover of G. That is to say, finding a maximum independent set is the same as finding a minimal vertex cover.

So this is pretty cool: generally, when we give transformations, the transformations are one-way: we typically map all instances of x to a smaller set of instances of the domain problem (in other terms, f is not surjective). However, because of the close correspondence of MIS and VC, we can show that they're equivalent in difficulty; that is, VC is NP-complete because it's the same problem as MIS—there is a bijective polytime transformation between the two (we're not quite done, but it's pretty easy).

17.4.5 Hamiltonian Circuit (HC) is NP-Complete

Given an undirected graph, is there a simple cycle that visits all of the vertices of the graph? It's easy to show HC is in NP: guess a permutation, and verify that it corresponds to a cycle. That is, for each pair of successive edges, check if the edge is present in G, and check to see whether the first vertex is equivalent to the last.

To show hardness, we'll use $VC \leq_P HC$. Following CLRS, we'll use the "exact" version of VC: instead of asking if a graph has a vertex cover of at most k, we'll instead use the DP "does G have a VC of size equal to k?". It's somewhat easy to see that if one is solvable in polynomial time, then so is the other. That is, it's equivalent to asking whether G has a vertex cover of size at most k, since adding vertices to a vertex cover yields another cover.

So we'll want to transform a VC instance $x = (G, k)$ to a graph G' such that G has a VC of size k iff G' has a Hamiltonian Circuit. We'll assume that G has no isolated vertices (they're not particularly interesting for the VC problem; they may, however, be slightly problematic for our proof as given).

We'll build a $G' = (V', E')$ that looks basically nothing like G; G' will be built up of little chunks, and have the property that $(G, k) \in$ VC iff $G' \in$ HC.

There are four parts to the construction:

1. design a "gadget" for each edge in E;

2. Construct gadget-gadget edges.

3. Add k "selector" vertices

4. Add gadget-selector edges.

So, eventually, we'll have $|V'| = 12|E| + k$ (the k vertices come from the selector vertices).

We'll have a strange little "chunk" for each $(u, v) \in E$ (of the input graph); each chunk has 12 vertices. I can't easily typeset the graph. so here's what it looks like. There are two vertical rows of 6, lined up next to each other. Call the LHS ones from top to bottom 0-5; ditto the RHS ones, 6-11. Connect 0-1, 1-2, ..., 4-5; ditto with the RHS. Further, connect 0-8 and 6-2; also, connect 3-11, and 9-5.

Now, note a Hamiltonian cycle has to hit all the edges of the gadget. Draw a picture and think about the following: suppose we enter, in our cycle, node 0. We can't follow the edge down to 8, because then we'd never hit 1. Therefore, after hitting 0, we have to continue on to node 3. Here, we have a choice: we can go down or up to 6. Suppose we go down—by similar logic to above, we have to go all the way down to 5, at which point we have to leave 5 and go somewhere else (covering the RHS vertices by some analogous path). On the other hand, at vertex 3, we could skip up to vertex 6. We thus must skip down to 8. At that point, we actually have to continue to 11, at which point we have to skip back up to 3, and down.

Thus, supposing we enter vertex 0, there are really only 2 things that can happen. Flipping the second route around, we have 3 ways we can traverse the gadget in a Hamiltonian cycle.

Second, now that we have gadgets for each $e \in E$, how do the gadget-gadget edges work? Take some $u \in V$ with degree d. Call the neighbors of u $v_1, \ldots, v_d$. Fore the gadget from (u, v_i) associate the LHS nodes with u and the RHS nodes with v_i. Consider the endpoints of, e.g., u's nodes. Line up all d gadgets for u, and connect the bottom node for each of the gadget parts with the top node for the next. Call the first terminal node for a u-gadget-side a superterminal; ditto the lat terminal node (the former will be at the top of some LHS, the latter at the bottom of some LHS).

Finally, we have $2|V|$ superterminals; add k selector vertices; connect each selector to all the superterminals.

Now, we show that this function has the property we desire for it to be a transformation.

First, suppose that $(G, k) \in$ VC. We will show that $G' \in$ HC. Suppose that $u_1, \ldots, u_k$ is a VC of G, numbered in an arbitrary way. We have selector vertices $s_1, \ldots, s_k$, and we'll have $|E|$ gadgets. We'll build a HC as follows. For s_1, for example, we'll follow it to u_1's superterminal. We'll proceed by following all the gadgets for edges incident on u_1—we can do so, because in our construction we connected all of u_1's terminals to each other. For a v_1 that's a neighbor to u_1, if it's not in the vertex cover, then we'll cover all 12 of the gadget's vertices; otherwise, we'll just pass through the LHS 6 of them. Finally, after traversing all of u_1's neighbors, continue back up to selector vertex s_2 and repeat. Finally, we'll traverse from s_k to all of u_k's nodes, and will eventually leave u_k's superterminal, ending up back at s_1.

On the other hand, suppose that $G' \in$ HC, for a G' constructed in the method above; we'll argue this implies that the original graph G has a vertex cover of size k. Fix a Hamiltonial cycle of G'. It must have two edges incident on s_1, as with every other vertex. Pick one of the outgoing edges. It can only go to one of the superterminals, by the construction of the selector nodes. Suppose it goes to the sueprterminal of u, for some edge gadget with u. No matter what happens, it will eventually go through the outoing terminal of the u side of that gadget—after that, if we don't go back to a selector node, then we must go to the terminal of some other u-gadget. Eventually, we will go back to s_2, at which point we will go to some superterminal (neither of the ones we saw before). That is to say, we'll end up at the superterminal of some other vertex v. We reason in the same way repeatedly until we have finally exhausted all of the selector vertices, at which point we must go back to s_1. Because of our construction, the Hamiltonian cycle must look like what we just described: since each s_i has degree 2, they must collectively be connected to $2k$ different superterminals. In other words, there must be k vertices $u_1, \ldots, u_k$ such that $s_1, \ldots, s_k$ are connected to the superterminals of $u_1, \ldots, u_k$.

Now the vertices $u_1, \ldots, u_k$ must form a VC; if it didn't, then there would be an entire gadget we didn't visit in the cycle (I don't quite get this step). The details of this proof may be found in the text.

18 Approximation Algorithms

18.1 LP Relaxation for weighted vertex cover

We consider a generalization of VC, called weighted vertex cover (WVC), where we care not only about the number of vertices in the cover, but also the total weight of the cover (with each vertex u having an associated weight $w_u \geq 0$). That is, given a vertex, weights, and a bound W, we ask if there is a vertex cover with weight at most W.

The optimization version is to compute a VC of min-weight. This is easily stated as a 0-1 ILP:

$$\min \sum_u w_u x_u \tag{47}$$

$$s.t. \quad \forall (u,v) \in E.\ x_u + x_v \geq 1 \tag{48}$$

$$\forall u.\ x_u \in \{0, 1\} \tag{49}$$

The x_u's are 0-1 variables, denoting whether we include node u in the vertex cover or not.

We saw that 0-1 ILP is NP-complete; we can perform the **LP Relaxation** trick: drop the integrality constraints, and instead require

$$\forall u.\ 0 \leq x_u \leq 1.$$

For the LP given above, we can throw away the upper-bound constraint, and simply require $x_u \geq 0$. Once x_u is equal to 1, it satisfies all the constraints that it's involved with; since we have a minimization problem, we certainly won't increase the value any more than that. Thus, the nonnegativity constraints suffice.

We may thus solve this LP to get the optimal fractional solution, and round to get the nearest integral solution (this general technique sometimes works, and sometimes doesn't). When we round, we send anything ≥ 0.5 to 1, and things < 0.5 to 0.

We must ask two things. First, is the rounded solution feasible—it's conceivable that in rounding we could violate one of the constraints. Note that before the rounding, the edge constraints were satisfied; at least one of the vertices must have had $x_u \geq 0.5$; if not, then the sum would be < 1. Thus, yes, rounding still puts us into the feasible region.

Second, we must ask ourselves how close to the optimal solution we will end up. The answer here is that rounding at most doubles the value of the objective function. The objective function is, recall, $\sum_u w_u x_u$. Suppose $x_u = 0.5$, then we at most double that term's contribution; if x_u is greater, then we less-than-double the objective function.

Since we solved the LP exactly, the value before rounding is the optimal fractional value of the objective function, which is $\leq$ the optimal integral value. So we must therefore also have that the solution is at most a factor of 2 from the optimal integral solution (by an argument I didn't quite follow).

18.2 A Primal-Dual 2-approximation algorithm for weighted vertex cover

We'll use the same LP relaxation we did in the last section, but we'll also form its dual, and consider an algorithm that not only updates the primal variables, but also alternately updates the dual variables; we'll work on both sets of variables together. We'll develop a method that doesn't require a general LP solver.

The primal LP-relaxed problem is, recall,

$$\min \sum_u w_u x_u \tag{50}$$

$$s.t. \quad \forall (u,v) \in E.\ x_u + x_v \geq 1 \tag{51}$$

$$\forall u.\ x_u \geq 0 \tag{52}$$

Then the dual is

$$\max \sum_{e \in E} y_e \tag{53}$$

$$s.t. \quad \forall u. \sum_{\{v:(u,v)\in E\}} y_{(u,v)} \leq w_u \tag{54}$$

$$\forall e.\ y_e \geq 0. \tag{55}$$

Now, the complementary slackness conditions say the following: Suppose you have a feasible solution to the primal and to the dual. Then, (1),

$$\forall u. x_u > 0 \Longrightarrow \sum_{\{v:(u,v)\in E\}} y_{(u,v)} = w_u$$

and, (2):

$$\forall e = (u, v) \in E.\ y_e > 0 \Longrightarrow x_u + x_v = 1.$$

In other words, they give conditions for tightness of bounds.

Now, we'll initially set $x = 0, y = 0$. If we keep x integral, we'll not be able to maintain optimality, because by complementary slackness (somehow, I got distracted), this will mean we've solved the ILP problem exactly, and that's NP-complete.

So we'll iteratively update x and y, while maintaining

- $\forall u. x_u \in \{0, 1\}$
- $\forall u. x_u = 1$ iff $\sum_{\{v:(u,v)\in E\}} y_{(u,v)} = w_u$. Note that, since we recall $x_u \in \{0, 1\}$, this is exactly equivalent to the first complementary slackness condition.
- y is a feasible solution to the dual.

Note that, initially, x is infeasible (assuming $E \neq \emptyset$). Also, note that the first two conditions are such that the dual variables y totally determine the x variables; it may thus be viewed as a program on dual variables (eh?).

So what we do is the following:

- While x is not feasible:
  - increase some y_e, maintaining the feasibility of y;
  - Update x accordingly.

How do we pick the y_e we want to increase? Well, let $e = (u, v)$ be an arbitrary edge in E such that both dual constraints (for u and for v) are not tight (thus, $x_u = x_v = 0$). We're guaranteed to have such an edge by an argument I was too busy typing to get. We then increase that e's y_e until at least one of the constraints (in the dual—either u's constraint or v's constraint) is tight.

Note that we didn't solve a linear program here.

Consider the value of the objective function upon termination. We argue that it's no more than twice the weight of an optimal min-weight vertex cover.

Now,

$$\begin{aligned}
\sum_u w_u x_u &= \sum_{\{u:x_u=1\}} w_u && (56)\\
&= \sum_{u:\text{dualconstr.istight}} w_u && (57)\\
&= \sum_{u:\text{dualconstr.istight}} \sum_{\{v:(u,v)\in E\}} y_{(u,v)} && (58)\\
&\le \sum_u \sum_{\{v:(u,v)\in E\}} y_{(u,v)} && (59)\\
&= 2\sum_e y_e && (60)\\
&\le 2 \times \text{opt} - \text{value} - \text{for} - \text{dual} && (61)
\end{aligned}$$

We have the last thing because y is feasible for the dual. So if y is a feasible solution for the dual, than its objective function $\sum_e y_e$ is $\le$ the optimal-value-for-the-dual.

By strong duality, we have that opt-value-for-dual is the same as the optimal-value-for-the-primal. That itself is less than the opt-value-for-integral-primal, that is, the primal problem with integrality constraints. So we have argued that the value of the primal objective is at most twice the optimum of the primal solution with integrality constraints.

18.3 Reductions are not Approximation-Preserving

So we've approximated, to within a constant factor of 2, the weighted vertex cover problem, and we've done so in polynomial time. Can we do better? Kind of. There is some $c > 1$ (1.36 or something) such that the problem cannot be approximated better than c unless $P = NP$. So coming up with a problem that makes an error of at most 10% on any input and runs in polytime is also NP-complete. Cool!

Note that this is a problem-specific trait. You can write down many different optimization versions of NP-complete problems, and they can behave quite differently with regard to approximability.

Recall, for example, that MIS and VC are exactly the same problem in terms of finding an exact solution (take the complement of one to get the other). However, they behave quite differently with regard to approximability. We may get, for example, a VC that contains all the vertices in the graph, and is 2-approximate; however, the complement of that is the empty-set, which is infinitely bad as a maximal independent set.

In other words, the standard polynomial-time reduction from one problem to another is not "approximation-preserving". We'd like to be able to compose two approximation-preserving reductions and get an approximation-preserving reductions, but the standard reduction just doesn't give us this (these complexity classes do exist, however).

MIS, in fact, cannot be approximated to within $n^{1/2-\varepsilon}$ unless P=NP (with n the number of vertices). So if you were to come up with an approximation algorithm that got MIS to within $\sqrt{n}$, you'd be able to exactly solve all NP-complete problems in polynomial time. (Further, under a stronger assumption, you show that you can't come within a $n^{1-\varepsilon}$ factor.)

18.4 (Fully)-Poly-time Approximation Schemes

We also have some NP-hard optimization problems that do better than above. There are some NP-hard optimization problems that admit a polynomial-time approximation scheme (PTAS)—we pick some ε, and we get a $(1+\varepsilon)$-approximation guarantee for all $\varepsilon > 0$.

Further, there are problems that admit a FPTAS: a fully-poly-time approximation scheme. What's that mean? Suppose we have a very small ϵ, and that the runtime of the algorithm is, say,

$$2^{1/\varepsilon}n^3.$$

For any fixed-epsilon, this is polytime, so it's a PTAS. However, it performs very badly for very small ε (we can construct much worse examples than this too). A FPTAS, on the other hand, would have running time polynomial in the input size n and in $1/\varepsilon$ (so

$$n^{10}(1/\varepsilon)^6$$

is an FPTAS.

For example, consider the knapsack problem. The decision version is, recall, NP-complete. However, this problem behaves very nicely with regard to approximability.

How? Assume, first, that we've thrown out all items that don't fit in the knapsack. Let v^* denote the max item value (thus, the optimal value is $\geq v^*$, since we assume everything fits in the knapsack). Now, we "round" the values up to the nearest integral multiple of $\Delta = v^*\varepsilon/n$. Intuitively, we're reducing the number of values we're considering. So v_i gives us

$$\tilde{v}_i = \Delta\lfloor v_i/\Delta\rfloor.$$

So $v_i \leq \tilde{v}_i < v_i + \Delta$.

How many values to we have to consider? Well, from v^*, we have $n/\varepsilon \cdot \Delta$ something, I have no idea what this expression means because fucking TeXShop decided to stop compiling and I've been trying to figure out why.

So each v_i is mapped to a $j\Delta$ for some $j \in \{0, \ldots, n/\varepsilon\}$. Any subset of items then has value $j\Delta$ for some $0 \leq j \leq n^2/\varepsilon$. This thus gives us that knapsack is a FPTAS, somehow.

What's the approximation guarantee? For each item, we can incur at most a Δ error; since there are n items, our total error is at most $n\Delta = \varepsilon v^* \leq \varepsilon \cdot OPT$.

So the optimal-rounded solution has its true value $\geq (1-\varepsilon)\cdot OPT$.

Now, the Traveling Salesman Problem (TSP) can be easily shown to be NP-complete by a reduction from Hamiltonian cycle. The Decision problem is to ask if there's a tour with a cost of a certain bound. So HC $\leq_P$ TSP: Suppose we have something to solve TSP; we set up a const matrix so that the cost of going from u to v is 0 if there's an edge from u to v in the graph, and 1 otherwise. We thus ask if there is a tour of cost 0, which is equivalent to finding a Hamiltonian Cycle. This problem has no PTAS.

A famous restricted version of TSP is Euclidean TSP (or metric TSP): in two dimensions, the points are points in the plane, and the costs are exactly equal to the Euclidean distance between the points. A PTAS has been given for the Euclidean version of TSP: for any constant-dimension and any constant ε, we get a polytime algorithm.

18.5 Log-factor approximation: weighted set cover

Further, there are NP-hard problems approximable to within a logarithmic factor (and no better, unless P=NP).

A canonical example of this is weighted set cover: this is a generalization of vertex cover. Suppose we have a set S of things we would like to cover (in VC, the set is the set of edges). We then have a set of sets $A_i \subseteq S$, and we want to find a least-cardinality (least-weight) collection of A_i's such that $\cup_i A_i = S$. (In the VC problem, these sets will be the set of vertices on an edge, I think.)

There is a simple $\ln |S|$ approximation algorithm for set cover (in fact, the approximation ratio is $\ln \max |A_i| \leq \ln |S|$). Further, there is a $c > 0$ such that an approximation algorithm for set cover achieving ratio $< c \log n$ entails that P=NP.

We'll give a pretty simple greedy algorithm for weighted set cover. We initialize our solution to the empty set. Then, we iteratively select a set A_i to add to our set cover, selected as follows. For a set A_i, we look at the w_i/N, with N the number of newly covered elements (that is, the number of elements in A_i that we didn't already have in our working set cover). Formally,

$$N = \left| A_i \setminus \bigcup_j A_j \right|$$

with j ranging over the sets we've already chosen. Intuitively, this is the cost per item newly covered. We thus pick a set A_i minimizing this ratio.

It's pretty clear that this runs in polynomial time—the total number of iterations is at most the number of sets we have.

We now argue our error is logarithmic. Suppose we have all w_i's = 1. Intuitively, suppose we have A_1 and A_2 each covering eactly half the points; suppose we have an A_3 covering slightly more than half; the optimal solution takes just A_1 and A_2; the greedy one takes A_3. Suppose there's now an A_4 containing slightly more than the size of $A_1 \setminus A_3$ or $A_2 \setminus A_3$; the greedy algorithm will select A_4. We can do this at most a logarithmic number of times (since the size of the $A_4, A_5, \ldots$ sets must halve at each iteration). So even if the weights are all 1, we can be off by a logarithmic factor.

We now give a slightly more general and formal proof that regardless of the weights, we won't be off by more than a log factor (that is, we give an upper bound on the error). We use a charging scheme. When we select A_e with ratio R, we charge $\$R$ to each newly covered element. At any point, the total amount of money at the elements will be the sum of the w_i's added. In particular, at the end, the total amount of money at the elements will be just the weight of the set cover.

The key **lemma** here will be: for any i, the total charge to the elements of A_i is $O(w_i \log |A_i|)$. Note A_i isn't necessarily a set in our chosen set cover; the claim holds instead for all the sets A_i. To see this, fix i arbitrarily. We reason about how much money can be in A_i. First, at the start of the algorithm, none of the elements have the money. Now, note that once an element receives money, it never receives money. So each element of A_i will have money placed on it at most once. That is, at each step of the iteration, zero or more of the elements in A_i will have money added to them, from some set A_j being added to the set cover.

Now, the first item of A_i covered cannot be charged more than

$$\frac{w_i}{|A_i|}.$$

Thinking about the computation of the ratio R when the first element was added, one of the sets considered for adding was A_i; the ratio for the set A_i at the beginning is $w_i/|A_i|$; if we chose set A_i, then the bound is tight. If, on the other hand, we chose a different set, then the ratio is smaller than that, giving us our inequality.

Consider now the second item covered in A_i. It could be the case that in the first round, some item is covered. Then the ratio of A_i in the round where the next elem from A_i is covered is

$$\frac{w_i}{|A_i| - 1}.$$

This is an upper bound, by similar reasoning to above: it could be that there is some set with a better ratio that's actuall chosen.

In general, the jth item that's chosen has cost

$$\leq \frac{w_i}{|A_i| - (j-1)}$$

(with the bound tight in the worst case, which happens when one element of A_i is covered each round).

So the total charge on A_i is less than

$$w_i \sum_{1 \leq j \leq |A_i|} \frac{1}{j} \approx w_i \ln |A_i|$$

since the harmonic series is approximately the natural log.

We now argue that the approximation isn't even logarithmic in the size of S, but instead logarithmic in the max cardinality of the A_i's. Now, let $A_{i1}, \dots, A_{i\ell}$ be an min-weight set cover; the total cost is

$$W = \sum_{1 \le j \le \ell} w_{ij}.$$

Now, optimal weight for A_{i1} is $\le w_i \log |A_i|$. Similarly, for A_{i2}, the total cost is $\le A_{1 \le j \le \ell}$ (total charge on A_{ij}). Note this is an overestimate in general: only in the case where A_{i1} and A_{i2} are disjoint do we get an equality here. Note the total charge on A_{ij} is

$$\sum_{1 \le j \le \ell} w_{ij} O(\log |A_{ij}|).$$

Let $m = \max_i |A_i|$; so we have $m \le |S|$. we then have the total charge on A_{ij} being

$$O(\log m) \cdot \sum_{1 \le j \le \ell} w_{ij}.$$

The sum to the right is exactly W; so we have our error is logarithmic in m.

An important inapproximability result: there is no poly-time $o(\log n)$-factor approximate algorithm for unweighted set cover unless $P = NP$. So to show $P = NP$, you don't need to exactly solve set-cover; all you need to do is give a sublogarithmic approximation. So the simple greedy algorithm presented is quite literally the best we could hope for.

18.6 Some Depressing Results about Boneheaded Randomized Approximation Algorithms

Consider **max cut**: we have a graph $G = (V, E)$. We want to color vertices either red or blue so that we have as many edges as possible going from the red side to the blue side. This algorithm is NP-hard.

A natural randomized algorithm is to pick colors at random. Let X be the number of bicolored edges. For each $e \in E$, define X_e to be the indicator variable equal to 1 if edge e is bicolored; 0 otherwise. Since $X = \sum_e X_e$, linearity of expectation tells us

$$E[X] = \sum_e E[X_e].$$

Recall that linearity of expectation always holds; the RVs above are not independent of each other: consider a triangle with three edges a, b, c; there's no way that all three can be bicolored. The dependencies/correlations are tough to pin down, but we still get linearity. Since $E[X_e] = 1/2$, then we have $E[X] = |E|/2$.

Now, it is known that max cut cannot be approximated to better than 17/16 unless P=NP (WTF?). There is an upper bound of something like 1.13 based on semidefinite programming.

Consider the MAX 3SAT formula. We're given a 3CNF formula; our goal is to find a truth assignment that satisfies as many clauses as possible. A natural random algorithm: flip a coin for each variable (set variables independently and uniformly at random). Recall that, in our 3-clauses, we require that the variables mentioned in the 3 literals all be disjoint. So an individual clause is satisfied with probability 7/8.

Let X be the number of clauses satisfied; for clause i define an indicator variable X_i which is 1 if clause i is satisfied and 0 otherwise. By linearity of expectation, we get

$$E[X] = \frac{7}{8}C$$

with C the number of clauses. So this algorithm, in expectation, gets you within 8/7 of the clauses. The hardness result here matches that: if you could get an algorithm that comes within $8/7 - \varepsilon$ for some $\varepsilon > 0$, then P=NP. Daaaaamn, son.

19 A Sketch of the Cook-Levin Theorem

The nice, formal version of this proof uses Turing machines. This sketch instead follows CLRS in giving a less formal outline, following the contours of the proof and ignoring bookkeeping matters.

We show that any language can be reduced to CIRCUIT-SAT, rather than the perhaps more traditional version that any language can be reduced to 3SAT.

How do we show a problem is NP-complete? Recall, L is NPC iff (1) $L \in$ NP; and (2) $\forall L' \in NP.\ L' \leq_P L$. (1) is quite easy (recall we just need to show that a problem solution is verifiable in polynomial time). (2) is where all the difficulty is.

Fix some arbitrary L in NP; we must argue that $L \leq_P$ CIRCUIT-SAT. The only thing we know about L is that it's in NP. Now, $L \in$ NP iff there is a polytime "verifier" algorithm $A(x, y)$ such that (1)

$$\forall x \in L.\ \exists y.\ |y| = O(|x|^c)$$

(c is some absolute constant) with $A(x, y) = 1$; and (2)

$$\forall x \notin L.\ \forall y.\ A(x, y) = 0.$$

The key idea here will be that we can simulate any polymonial-time computation via a polynomial-sized circuit. That is, we need a polynomial-time f mapping any string x to a circuit $f(x)$ such that $x \in L$ iff $f(x) \in$ CIRCUIT-SAT.

We don't think about Turing machines, but instead in the abstract RAM model of serial computation we know and love. We have memory, we have registers, control logic, and so on; in other words, we have bits of state. At every clock tick, the state gets updated by passing through some sets of combinational circuits—two registers may be added together, for example, depending on the value of certain control bits. In other words, each clock tick takes the memory vector and outputs a new memory vector. Abstractly, we think of our machine as a large combinational circuit that, every clock tick, takes the output of itself and feeds it into the next step.

In other words, if we run the machine for k steps, and we want to transform the input to the output after k steps, we imagine just copying the hardware k times and wiring it together. If our computation runs for a polynomial number of steps, we're not blowing up the normal size of the combinational circuit by more than a polytime factor.

In particular, we have to examine what happens to the verifier on a specific given x. The verifier algorithm takes inputs x and y; when it terminates, there is some particular position in memory to which the output is written. x is a fixed sequence of 0s and 1s given to us; we treat the y inputs, however, as variables. For simplicity, imagine all the other memory locations are zeroed out.

We do the thing where we replicate the combinational circuit implementing A a bunch of times; we can prune away the circuit to a minimal one that only has y as an input (that is, propagate the info from x upwards); we can trim away all gates that don't enter into the output bit too.