18.415 Advanced Algorithms

Notes from Fall 2020.

Last Updated: November 27, 2020.

Contents

Introduction

These notes were typeset live from the online 18.415 lectures, taught by David Karger.

Remark. *Though the course was originally designed as 'Advanced Algorithms,' it is probably better to refer to it as 'Classical Algorithms.'*

The course doesn't go into any cutting-edge algorithms; rather, it provides us a toolkit for algorithms that we are expected to know. We'll mostly be looking at combinatorical problems, and introducing various models to be able to understand them as well as their efficiency.

Most course information can be found at the following link: `courses.csail.mit.edu/6.854`. The core of the course are long and challenging PSETs, and is complemented by a peer grading assignment and a final project. Collaboration is encouraged and essential, though they should be small groups of size 3. Academic integrity is critical to the course, though in recent years people have always been caught.

Due to current conditions, there are many experimental tactics in regards to lectures. Lectures will be recorded, though live lectures will allow for questions. Playing previous lectures may also happen, in where we can stop and ask questions. Self-study / giant office hours during lecture may also be possible. Problem set structure may also be changed, due to lack of facility of collaboration. Collaborators will be changed throughout the semester. There will be sufficient 'slack points' for late PSET submissions, but extensions will be granted as well if necessary.

1 Lecture 1: Fibonacci Heaps

1.1 MST problem review

Fibonacci Heaps were developed by Fredman and Tarjan in 1985, and were done so specifically to solve the Shortest Path/Minimum Spanning Tree problems. Previously (in 18.410) two algorithms for solving MST problems were introduced: **Prim's algorithm** and **Kruskal's algorithm**. Both of these algorithms are greedy algorithms, with Prim's relying on building up the tree one by one, adding the closest neighbor to the tree. On the other hand, Kruskal's builds up the tree through connecting different components, adding in the shortest useful edge that connects two components.

There is actually another, 'simpler' algorithm that is taught much less often to solve the MST problem: **Boruvka's algorithm**, where we **contract** a minimum edge off of each vertex. It is somewhat similar to Kruskal's algorithm. It starts off by putting every vertex in its own component. At every iteration, the minimum weight edge out of every component, that connects to a different component, is added to the MST. Since each iteration reduces the number of components by a factor of ≥ 2, this algorithm takes $O(E \log V)$ time.

When we implement Prim's algorithm, we want to use a priority queue, where we are able to insert items, find/delete the minimum element, and decreasing the key (updating edge vertices). Prim's algorithm is actually isomorphic to Dijkstra's algorithm; the only difference is the cost function. The key is the distance to the source in Dijkstra's, while it is the distance to the root in Prim's.

What is the cost of Prim's algorithm, in terms of n vertices and m edges? Let's make a quick table for everything when we use a heap as a priority queue:

operation	number of operations	runtime per operation
insert items	n	$O(\log n)$
find/delete minimum	n	$O(\log n)$
decrease key	m	$O(\log n)$

The total runtime is $O((m+n)\log n)$.

Note 1.1

Heaps should have been covered in standard undergraduate algorithms courses. As a review, a **heap** is a tree that satisfies the **heap property**, which states that if P is a parent node of C, then the key of P satisfies some relation compared to the key of C. In a **max heap**, that relation is $\geq$; in a **min heap**, that relation is $\leq$. A standard heap uses $O(\log n)$ amortized time for most operations.

1.2 Using d-heaps for MST

Since the decrease key operation is the most expensive operation of our algorithm, and so any modification to decrease this runtime would help us. This brings us to our key principle of **balancing**, which is necessary for our heap structure. So, instead of using a standard heap, we would instead use a **d-heap**, which is a heap with d children per node, with height $\log_d(n)$. This brings the insert/decrease runtime down to $O(\log_d n)$, while the delete min operation requires time $O(d \log_d n)$, ultimately changing the runtime to $O(m \log_d n + nd \log_d n)$.

Question 1.2. *How do we minimize runtime?*

Claim 1.3

If we set $m \log_d n = nd \log_d n$, then we will get a factor of 2 within the minimum.

Proof. Exercise. $\square$

Either way, setting them equal ($m = nd$), the runtime becomes $O(m \log_{m/n} n)$. This doesn't seem like a big improvement, but in a dense graph, this actually makes it a linear time algorithm!

1.3 Introduction to Fibonacci Heaps

Can we do better? The answer is yes, with Fibonacci heaps.

> "Fibonacci Heaps are an excellent demonstration of laziness. You should never work unless you have to, since there is too much work to do. Similarly, you should always procrastinate. If you start early, then you end up doing work that potentially may be never needed."
>
> -D. Karger

The core idea of data structures is to put off work until later. When you're forced to work, make it count, not only to answer the question, but also to simplify the structure itself. A good model is to use an adversary, which tries to make you work, and we try to optimize and amortize our work versus theirs, which we formalize with a potential function.

So, let's analyze our 'Fibonacci heap' (the name will become clear at the end of the analysis).

How do we be lazy with regard to insertion? We can use a linked list, and we can push with $O(1)$ time in this way.

Joke 1.4. *A more time efficient way to do this, is simply to say "ok," to a request for insertion. This uses* 0 *time, but obviously poses a problem in that every subsequent non-insertion operation will return the wrong answer.*

How about delete min? We need to scan for the minimum, which takes $O(n)$ time for n elements. But amortized, this is only $O(1)$ per inserted item! This is actually faster than the priority queue, which takes $O(n \log n)$ time, since the heap property has to be maintained. But obviously we have a problem if a bunch of delete min operations are asked in succession, in where it will take $O(n^2)$ time asymptotically.

So instead, we'll go back to our original principle of trying to simplify the data structure. Each comparison that we make in finding the minimum element also reveals to us information about smaller/larger elements. We can use this information to simplify runtime - simply ignore the larger elements when looking for the minimum element. To implement this, we'll make a tree in where we compare every element, and the larger element in the comparison is made the child of the smaller element. This ends up making a **heap-ordered tree (HoT)** (also known as a **caterpillar**)! Now, the minimum is at the root after creating this tree structure. After the first delete-min operation, only roots of subtrees are possible candidates for the next delete-min operation! This is a big time improvement, making the runtime proportional to the maximum number of children of the root.

But this is exactly the problem - our data structure is too inefficient when the root has many children. If we start comparing the elements starting with the minimum, then we just get a tree of height 2, with everything connected to the root minimum. After deleting the minimum node, then this just turns into $n - 1$ isolated elements, and so we lose all the information that we obtained regarding comparisons. Our previous approach put too much information on the minimum, which is lost after deletion, and hence we want to limit the number of children/node. To solve this, we can instead arrange a competition, in bracket style (similar to a binary tree). In this scheme, we compare random pairs of elements, then compare the smaller ones of those pairs, then compare the smaller ones of those, and so on and so forth. After rearranging everything then, we end up getting a tree known as a **binomial tree**. This is because there are $\binom{\log_2(n)}{k}$ elements at every tree depth k, where n is the number of nodes. This also implies that deleting the minimum will only produce $\binom{\log_2(n)}{1} = \log_2(n)$ subtrees.

This binomial tree structure still does not solve all our problems, since insertions and delete-mins can be interleaved, and we can't run this bracket system each time. To get around this, we'll instead record each node's degree, which is equal to the number of children that a node has, or in other words, which 'tournament round' it is in (for example, a node that lost the first 'round' would have a degree of zero, while a node that won two rounds would have degree 2).

After delete-min is run, we will end up with a number of new HoTs, equal to the degree of the previous minimum. These will all be added to our collection of HoTs. Afterwards, we'll only face off HoTs whose roots have the same degrees (known as the **union by rank heuristic**, similar to the union-find data structure). To consolidate them through this heuristic, we will append HoTs of degree d with each other, by connecting the root of one tree to be a child of the other. This reduces two HoTs of degree d to one of degree $d+1$. This process starts from the HoTs with root degree 0, in order to end up with only one HoT of each root degree. By a similar analysis of the union-by-rank heuristic as the one in union find, the maximum degree we have is $O(\log n)$ and hence the total number of HoTs we have is also $O(\log n)$. This collection of HoTs is what we know as the Fibonacci heap.

If we combine this with lazy insert, we'll have insertion just add a degree 0 HoT to the collection instead. On a delete-min, we will first delete the minimum element, and then put its children into the collection of HoTs. We then consolidate to find the next minimum element, simultaneously shrinking down to only having $O(\log n)$ HoTs.

Question 1.5. *What is the runtime of the delete-min operation?*

Theorem 1.6

In the worst case, the runtime of the Fibonacci heap delete-min is just $O(n)$, which happens when there are n inserts (making n degree 0 HoTs) followed by a delete-min. This operation is amortized $O(1)$ due to the insertions, but also allows for restructuring of the entire data structure to be a Fibonacci heap. Afterwards, the runtime for delete-min is proportional to the number of HoTs we have, plus the number of new trees created by deleting the root, which is $O(\log n)$.

Proof. At the beginning, when there are n HoTs of degree 0, each node needs to be compared to build up the original Fibonacci heap, take $O(n)$ time. Afterwards, due to the structure of the Fibonacci heap, a deletion only requires consolidation of the remaining HoTs. There are at most $O(\log n)$ HoTs originally, plus the number of children we have (bounded by the binomial heap structure as $O(\log n)$, and so consolidation will take at most that much time. □

To formalize this, we can go through an amortized analysis with a potential function.

Note 1.7

As a review, in the potential method for amortized analysis, we have that

$$a_i = r_i + \Phi(i) - \Phi(i-1)$$

where $\Phi(i)$ is our potential function, r_i is the real cost, and a_i is the amortized cost.

We have that $\sum a_i = \sum r_i + \Phi(\text{final}) - \Phi(0)$ by telescoping. If $\Phi(\text{final}) \geq \Phi(0)$ then the sum of amortized costs $\sum a_i$ is greater than or equal to the sum of the real costs $\sum r_i$, and hence the amortized cost becomes a upper bound on the actual cost. If the potential is always positive and $\Phi(0) = 0$ then this condition is automatically satisfied.

For our case here, we can let $\Phi(i)$ simply be equal to the number of HoTs we have. Then, the amortized cost a_i will equal the final number of HoTs we have, plus the number of children that we make $O(\log n)$. Since the number of HoTs after a delete-min operation is $\log n$, the final amortized cost is simply $O(\log n)$!!!

Hence, such a data structure provides $O(1)$ insertion and $O(\log n)$ amortized delete-min. But this is not good enough for us, since we're still missing the decrease-key operation. This will be covered in depth in the next lecture, but the algorithm is simple - just cut off that subtree and make it a new HoT. The problem is that we can make trees that are not binomial trees, which is a problem due to their size (the number of children).

Remark 1.8. *Usually, the location of the minimum element is also marked in the delete-min operation, such that accessing the minimum will take $O(1)$ time rather than the $O(\log n)$ time required to look through the tree roots we have in Fibonacci heaps.*

2 Lecture 2: Fibonacci Heaps, Persistent Data Structs

2.1 Fibonacci Heaps Continued

We start off today with a review of how we defined the Fibonacci tree, that we defined last time with our collection of HoTs. We had the potential function being equal to the number of trees we had, and since the number of trees is $O(\log n)$, our amortized cost for delete-min is $O(\log n)$. Finally, we closed off by mentioning the decrease-key operation, which was simply to cut off the tree that decreases-key. However, this has the problem that if a node loses children over and over again, we no longer have our $\log n)$ bound.

Remark 2.1. *Prof. Karger mentions the movie "Saving Private Ryan," in where four children went to fight in war. The family ultimately decided that it was best to rescue the last of the children, since it would be horrible if a family lost all their children.*

And so we can do the same with the Fibonacci tree, to fix our decrease-key operation. "If a node loses more than 1 child, then we will take it out of the line of fire" (i.e. make it a root node and cut it off as well) To implement this, we will add a "mark bit" to each tree that represents if a child has already been cut from a node. Then, a parent of the key which decrease-key is run on will be cut from the tree if it was marked. This has the potential to cause a cascading cut, in where we keep cutting until we find a clear mark bit.

Theorem 2.2

We will have a nice data structure with $O(1)$ insertion and decrease-key and $O(\log n)$ for delete-min, if we can prove the following:

1. Show that cascading cuts are "free" amortized.
2. Show that the tree size will be exponential in the degree of the root. The base of the exponent doesn't matter, since a size of n will correspond to a degree of $O(\log_b n) = O(\log n)$.

Proof. **Claim 1** To show the cascading cuts are free, we will make a potential function for the marks, namely Φ = number of marked nodes, in all our trees in the data structure. Now, what is the cost of the decrease-key operation? The cost is just simply equal to the number of cut nodes c, plus $O(1)$ for 'housekeeping operations.' The number of marked nodes decreases by $c - 1$, plus the one new node we mark, for a total change in potential of $2 - c$. This means that the amortized cost is constant $O(1)$!

But this has a problem. The potential function differs from the previous potential function, and that's cheating, since we're making a bunch of new HoTs in our Fibonacci heap as well. We need to reconcile these potentials, and so we define the final potential to be $\Phi =$ **(number of HoTs) + 2·(mark bits)**. The reason we have a factor of two in front is so we have one unit of work for the cascading cut itself, and the other unit of work to account for the increase in the number of HoTs. Going through with this analysis, we will then see that decrease-key will have amortized $O(1)$ time.

Claim 2 To show a bound in the tree size, we'll consider a node x and its current children $y_1, y_2, \cdots, y_k$, where each y_i is ordered by order of addition.

Lemma 2.3

y_i has degree $\geq i-2$

Proof. Consider the arbitrary child y_i. It was added after children $y_1, y_2, \cdots, y_{i-1}$. When y_i is added, then x itself had degree $\geq i-1$ since the other y_i are children. Since we only consolidate nodes of same degree, then the degree of $y_i \geq i-1$ when it was added. Because y_i is still in the tree, then due to our rule for cutting, at most one child could have been cut from y_i. This means that the degree is $\geq i-2$. □

Now that we have this bound on degree, we can figure out how big the tree itself is. If we let s_k be the minimum number of descendants (including children) of any subtree with a degree k root, then we aim to find a bound on s_k itself. We can easily find $s_0 = 1$ and $s_1 = 2$. We further have that $s_k \geq \sum_{i=2}^{k} s_{i-2} = \sum_{i=0}^{k-2} s_k$.

Solving this at inequality at equality, we get that $s_k - s_{k-1} = s_{k-2}$, and we see that rearranging it shows that s_k satisfies the Fibonacci recurrence! So $s_k = O(\phi^k)$ and hence is exponential in the degree of the tree. □

So, we see that fib-heaps are pretty useful. We have $O(1)$ insertion and decrease-key, and $O(\log n)$ delete-min. We also have $O(1)$ merging. This is actually optimal, since otherwise we could get a comparison sort with time $\Omega(n \log n)$, which is impossible.

Is it practical? Well, the constants of the operations aren't bad, and sometimes they outperform binary heaps. However, fib heaps use pointers to store everything, rather than arrays (in **implicit heaps**). Additionally, CPUs also use caching, and memory-accesses are free in arrays rather than all over the place pointers, and so we lose caching efficiency if we stick with Fibonacci heaps.

2.2 Applications of Fibonacci Heaps

Recall that Fibonacci heaps were originally designed to solve the Prim/Dijkstra shortest path-finding/minimum-spanning tree problems, and a regular heap could get time efficiency $O(m \log n)$. With a Fibonacci heap, the decrease-key operations are all $O(1)$, and so the runtime improves to $O(m + n \log n)$, and this is linear except when the graph is really sparse.

Can we do better, even for sparse graphs, trying to get rid of the logarithmic term? Let's look at MSTs to start with - they're still slow with Fibonacci heaps, since n delete-mins from a size n heap will take $O(n \log n)$ time. To get around this, we should try to keep the heap size small, say less than a constant k.

We'll start of with a standard Prim's algorithm: start from a node, and insert neighbors, delete min from the heap, etc, until the heap size hits k. The heap itself contains the neighbors of the current tree, and we will keep running the algorithm until we have $\geq k$ potential neighbors. To actually find the MST then, we will go to another node that has not yet been added to the MST, and start running Prim's on that one, ending with the same condition or when we connect to a previously established tree.

After one iteration of this procedure, we end up with many subtrees that have yet to be connected to each other. This takes $O(m + t \log k)$ time (through a Fib heap) where t is the number of nodes we have by standard Prim's algorithm. To combine all the trees, we can contract each created subtree to a node (in $O(m)$ time), and restart the MST algorithm with a new k (this will be known as one phase). Then, if we make $t \log k = m \implies k = 2^{m/t}$, we get linear time in m per phase.

Now, how does these constants change at the conclusion of every phase? Since each contracted node will have $\geq k$ incident edges, the number of vertices at the after contraction at the end of every phase will be $\leq \frac{m}{k}$. This means that the number of nodes in the next phase is going to be $\leq \frac{m}{k}$, and so to achieve linear time per phase, the new k of the next phase can be $2^{m/t'} \geq 2^k$.

Initially, we start with $t = n$ by definition, and with $k = \max\left(\frac{m}{n}, 2\right)$ (so we don't start with a crazy small k). What is our number of phases? Well, since by definition we will have an MST if $k = n$, then we can simply look at the time it takes for k to reach n. With every phase, the number k gets *exponentially larger*. We call this number of phases $\beta(m, n) = \min\{i \mid \log_2^{(i)}(n) \leq k_0 = m/n\}$, and it is easy to show that $\beta(m, n) \leq \log^*(n)$. $\log^*(N) \approx 8$. where N is the number of atoms in the universe. This is essentially linear.

2.2.1 Further Improvements

But Theoreticians are never satisfied. This bound was improved to $m \cdot \log \beta(m, n)$ using edge bundles. This was further improved by Chazelle to $O(m\alpha(n) \log \alpha(n))$ where $\alpha(n)$ is the inverse Ackermann Function, where $\alpha(N) \approx 4$.

We can get even better with randomization, into time $O(m)$. It is unknown if this is able to be solved in linear time yet, but surprisingly (!!!) we can actually get an optimal deterministic algorithm for solving MSTs with unknown runtime. (proof by Pettie and Ramachandran: brute force all algorithms for tiny spanning trees, choose the minimum, and the exhaustive search for best algorithm takes linear time. Then generalize to larger trees.)

2.3 Introduction to Persistent Data Structures

Tarjan is basically the God of data structures. He is also one of the best writers of theory papers. He and Sarnak also introduced the idea of persistent data structures. The general idea is to augment the data structure to support operations in past versions as well. (for example, to answer questions like: "what was the min yesterday?") The ability to query the requisite task is known as **partial persistence**, but they even provide an ability to "insert a new key in the heap yesterday" and ask "what was the min 3 hours ago?" incorporating the insertion of the key yesterday as well. This is basically time travel, called **full persistence**, where we can query and modify the past.

They came up with a way to do this, without regard to the type of data structure that is used. Any structure that is pointer based, which has fixed-size nodes whose fields hold values and pointers to other nodes, can be augmented. (Glaring Exception: no arrays)

Now, the idea is to wrap the data structure and the primitive operations on it. When modifying, we change the value or pointer in a node itself. Thus, if we can figure out how to make a node persistent, then we can extend this to the entire data structure. Changing each field to an array of time/value pairs (the **fat-node method**) allows us to do this. We pay a $O(1)$ space cost per insertion. The runtime of insertion is still $O(1)$, however queries become longer, to $O(\log t)$ time with binary search. This is a multiplicative slowdown, unacceptable for theoreticians, which we will resolve in the next lecture.

3 Lecture 3: Persistent Data Structs, Splay Trees

3.1 Persistent Data Structures Continued

Recall from the last lecture that data structures are mostly pointer based structures, which are collections of $O(1)$ size nodes that contain $O(1)$ scalar values, as well as $O(1)$ pointers to other nodes. We want to make what we call **atomic operations**, query and modify operations, persistent; doing this for every node makes the whole data structure persistent. We previously also talked about the **fat-node method**, which was unacceptable due to the multiplicative $O(\log t)$ slowdown for lookup (need to do this for every node).

To start off in looking at how we can implement this, let's consider a binary search tree, when we try to add a node to it. If we just overwrite the entire binary tree, then we lose persistence, even partial persistence. A brute force approach would result in storing a copy of the binary tree at every time step, the **copy method**, which is unfortunately quite slow. The lookup time is $O(\log t)$, where this t is now the total number of modifications we ever made. But the good thing is that this is an additive $\log t$ term, since we only need to find the root (rather than for every node), and then we have a data structure to work with. The problem with this approach is that a modify operation is $O(n)$ and also takes $O(n)$ space, which is highly unacceptable.

How do we get around this? The key idea is that *almost the whole tree is the same after each operation.* This means that if we just keep track of changes, rather than the whole tree at each time t, then we can improve our modify operation. We copy the new node, as well as everything that changed because of this addition. The parent of the new node changes (it now has a pointer to the new root), and the parent of that also changes (since its child now has an additional pointer), and so on. So, if we make new nodes corresponding to the ones that need to be modified, with *pointers to the old unmodified nodes*, then we maintain persistency with less time and space (still unbounded: $O(\text{number of ancestors})$). This method is known as **path copying**.

Tarjan found a way to combine the path-copying $O(\log t)$ additive slowdown with the $O(1)$ modification in fat-nodes - to consider "plump nodes" instead. The idea is that fat nodes are bad, and so if it's too fat, then we do path-copy instead.

We'll only go over this idea for tree structures, but it is not too hard to generalize to all pointer-based data structures (one can see this in the original Tarjan paper). In each node, we will have one extra timestamped field. We'll use it to store the timestamp and action of the first modification (even changes in pointers) of the node, and on the second modification we make a copy of this node (with a new empty extra field). Once the second modification is made, then we will change the time-stamped field of the parent to store the fact that its child changed to the new copied node at that time t. Thus, we will have two options when we traverse to this node, depending on the time t - we will either point to the original child, or to the new node (this information is completely stored in the extra field, so no information is lost). Then, at a later time, when we want to modify that node, we will simply modify the new copy instead.

How do we implement the lookup operation? We can start at the root of the tree, and check if the field has a timestamp of modification before or after time t. If it was, then we go to the new node as specified by the extra field. Otherwise, we go to the previous node. This is the operation of looking up a particular node, and since we only add a constant amount of operations, this is a $O(1)$ slowdown. But there is a slight problem, because we still need an array of roots ordered by time. Every time the root is modified, then there is no parent that can store that change, and so we need to make a copy of the node itself. As a result, there is an additive $O(\log t)$ slowdown since we need to find the appropriate root to start with.

What about the new space that this uses? Let's do an amortized analysis, with the potential being equal to the number of **live nodes** that have a non-empty extra field. This is only a problem for nodes that we may still modify, and we aren't going to modify nodes back in time. Therefore, we define a **live node** to simply be nodes reachable from the current root.

Let's calculate the amortized space cost for an atomic modification. If we modify a node and the extra field is empty, then we can simply do this with a constant space increase, and the potential increases by 1. If we modify a node with a non-empty extra field, then we copy it, point the parent to the new copy, and then the old node becomes 'dead,' resulting in a decrease of 1 in the potential. Then, since the copy operation increases by $O(1)$ space but the potential decreases by 1, then this is a free operation! These costs are going to cascade because it ends once we get to a node with an empty field, so we just get an overall space cost of $O(1)$ for modification.

3.2 The Planar Point Location Problem

This is a problem in **computational geometry**, which is a field we will return to later in the course, but it can be solved with a persistent data structure approach!

Definition 3.1

Computational geometry involves problems dealing with points and lines in the plane or in space, and ignores the algebra for computation. Instead, we focus on geometrical concepts, and have operations such as checking for line intersections, checking if points are on lines or not, and the length of the segment itself, as $O(1)$ primitive operations.

Definition 3.2

The **Planar Point Location** problem is where we have a planar subdivision, which is a splitting of the plane into a bunch of polygons. The question is to determine which polygon is a certain query point in. Line segments can emanate from any vertex, and can also end at any vertex, and the input size is in n, the number of segments.

The 1D version is simple - it is simply to ask which segment a query point itself lies on. We can easily solve this with binary search, and a binary search with sorted list of the endpoints can easily answer each query in $O(\log n)$ time, with $O(n \log n)$ time and $O(n)$ space to build.

A trick that's often applied in computational geometry is **dimensionality reduction**, where we try to change a 2D problem into a 1D problem. We'll use it here. If we project all the vertices of polygonal intersection onto a line, then we know which "slab" (x-coordinate region) the query point is in. We can then binary-search the segments, asking if the query point is above or below a specific line segment. The reason that the slabs are important is that it removes the ambiguity of above/below, and restricting it to just one slab results in no ambiguity and a total ordering.

What is the cost of this? We have two costs - a build cost and a query cost. For the query cost, we can find out which slab we are in in $O(\log n)$. After that, we can find the y region that we are in also in $O(\log n)$. Therefore, we can find the query cost in $O(\log n)$ with two binary searches in 2D. This generalizes to n dimensions easily, becoming $O(d \log n)$.

For the build cost, let's first consider the space cost. The number of slabs is proportional to the number of the vertices, and for every slab we need to keep track of the ordering of the segments, resulting in a total space cost proportional to $O(\text{vertices} \times \text{segments})$. And this can easily become $O(n^2)$, which is bad - this also means that our runtime is at least $O(n^2)$, unacceptable.

To solve this, we'll use persistent data structures instead. Instead of considering the standard xy plane, we'll instead consider the x axis as a measure of time t, and keep track of a vertical line that moves from left to right (this is known as the **sweep line** method). Our question is to find out what the sweep line sees as time progresses. It sees intersections with some line segments, and these intersections do change coordinates as the sweep line moves to the right. But we don't care about numbers, so we don't need to keep track of them. The only thing that actually needs to be taken care of are changes in the ordering of the segments, as well as the appearing/disappearing of segments themselves. Segments can only appear/disappear/cross at vertices themselves, and so the only topological changes we need to keep track of are precisely at the x coordinates of the vertices! Each change is relatively small, and so we can keep a persistent binary search tree on the segments in the sweep line.

These modifications of addition/deletion/crossing are all cheap with persistent data structures. We can use a balanced red-black tree, or more generally, any balanced binary search tree (BST). Each update to the persistent BST will only take $O(\log n)$ time (as is standard for a red/black tree), and a $O(\log n)$ space change (due to data structure persistency). As we sweep the line across the data structure, we will at most be doing $O(\text{segments}) = O(n)$ changes, and so all our insertions and deletions will just be $O(n \log n)$ time and space!

With this implementation, when we get a query, we can just query a slab by looking at the by 'time,' the x coordinate. Lookup is going to be $O(\log n)$ as discussed earlier with persistent data structures, and so we have the same query cost and the same building time cost as the 1D case. Space is still $O(n \log n)$, but we can refine this by looking at how red-black trees work. They modify $O(\log n)$ bits at each operation, but only do one rotation. We don't care about the historical red-black bits, since they don't matter for anything except for rebalancing. We'll only persist the rotations, and since there's only one per insert/delete operation, this improves the space usage to $O(1)$ per operation. Therefore, the space used by our red-black tree without persistently storing the color bits is simply $O(n)$, exactly equal to the 1D case. These time and space bounds are optimal!

3.3 Introduction to Splay Trees

Splay Trees, invented by Sleator and of course Tarjan, are another type of balanced binary search tree, which maintain $O(\log n)$ operations. There have been many balanced BSTs developed: red-black, AVL, scapegoat, 2-3 trees, etc. But all of these previous trees are kind of annoying, since you have the hassle of tracking extra information. Sleator and Tarjan developed self-adjusting trees, which don't store any balance info, and are "absolutely astounding".

These self adjusting BSTs don't have any balance info, and are sometimes not even balanced, but nevertheless take $O(\log n)$ amortized time. They even outperform regular BSTs in formalizable ways, and contain more operations than a regular BST. They can be merged and split in $O(\log n)$, and other more complicated operations can even be free.

The drawback of splay trees? A much more sophisticated analysis, involving "cleverness and black magic." The idea is to use a potential function to measure imbalance, and you need many insertions to cause imbalance. However, searches will decrease the potential, and so are paid for even if there is an imbalance. We'll elaborate on this for the next lecture.

4 Lecture 4: Splay Trees

We know how binary trees work - they will have all operations proportional to the depth, which is usually $O(\log n)$ but can be as much as $O(n)$, unless we use some sort of balancing. But splay trees are special. They don't store any auxillary information, and yet are still able to be provably balanced. We'll have a potential function Φ which measures how unbalanced the tree is, in order to make any operation amortized $O(\log n)$. Splay trees, like the data structures we've seen so far, are efficient because they are lazy.

4.1 Heuristics of Splay Trees

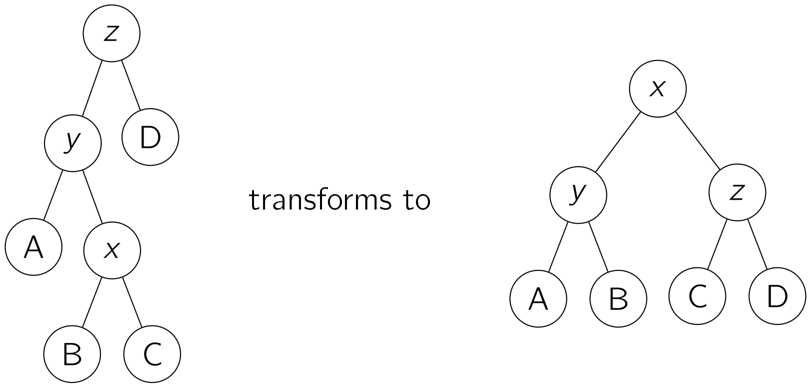
One of the primary heuristics in making the splay tree is by shortening long paths. Long paths take a long time to traverse and have a large potential, and shortening them reduces the potential and pays for the traversal itself. When we insert, the path increases, and the potential also increases such that we have enough to pay for a search later.

Another heuristic is to rebalance with rotations. We'll say that a tree is balanced if the left and right subtrees of the root roughly have the same size. During a search, if the tree is balanced, then searches are fast and searches can be $O(\log n)$ time. In general, if the subtree is less than some constant factor of its parents' size, then the tree is balanced.

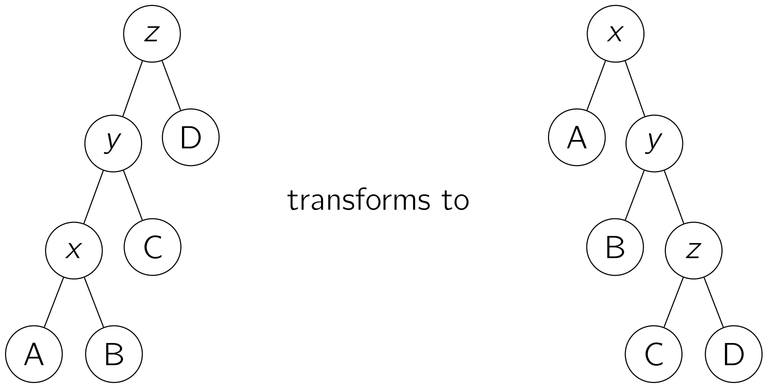
4.2 Solving the Balancing Problem

The problem with trees appears when we have fat children. Then, when we descend, a search through a fat child takes significant time, so we'll say that fat children have a large potential. Thus, to reduce the time of operations, we should eliminate fat children. One proposal to do this is to rebalance through rotations, but this doesn't work, since rotations may still leave behind a long path.

What does work, however, is a double rotation. These double rotations will take different cases, based on the relative positions of the wanted node, their parent, and their grandparent. We'll define a **zig-zag** operation of a node x to take place if x is the right child of its parent y, which is the left child of the grandparent z. The zig-zag operation will involve first rotating x with y, and then rotating x with z. The transformation is shown below:



We'll also define a **zig-zig** operation to take place if x is the left child of its parent y, which is the left child of the grandparent z. Here, we'll first perform a rotation between the y and z, and then x and the now-parent z. Finally, a **zag-zag** operation is the symmetric variant of the zig-zig operation, except with everything reversed, and takes place when x is the right child of y, which is the right child of z. The zig-zig operation is shown below:



Where did the intuition for thinking double rotations may work come from? Prof. Karger doesn't really know himself:

> "This is what separates the truly great. A few lightning strikes of insight are enough to get you into the textbooks."

4.3 Implementation and Analysis of Operations

We now define the **splay** operation of a node x to effect a double rotation of the node up the tree, until it becomes the root node. If x eventually becomes the root's child, then we just do a single rotation at the end.

With the splay operation, we define the **search** operation to simply find the node x with standard techniques. Afterwards, we splay x, bringing it to the root. This is $O(\log n)$.

For our analysis, we will assume that each item x has a weight w_x, which is not intrinsic to the data structure, but will help with the analysis. For now, we'll set all the weights w_x to be equal to 1, and we'll additionally define the **size** function $s(x)$ of a node x to give us the total weight of the subtrees of x. For example, when all the weights w_x are equal to 1, the size function simply counts the number of nodes of the subtrees rooted at x. We will define the **rank** function $r(x)$ to be $\log_2 s(x)$, which intuitively gives us the 'height' of the splay tree. Finally, we define the potential Φ to be the sum of all ranks of all the nodes in the tree. Intuitively, Φ highly penalizes large deep subtrees, and rotations/splays to raise the subtree itself will help us raise the potential.

We will now introduce a key lemma in our analysis:

Lemma 4.1 (Access Lemma)
The amortized time to splay a node x given the root t is equal to

$$3(r(t) - r(x)) + 1 = O(\log(s(t)/s(x))),$$

where the functions r and s refer to the initial positions of x and t. We will also define a function $r'(x)$ to refer to the rank of x after splaying, and so the Access Lemma can also be written as follows:

$$3(r'(x) - r(x)) + 1 = O(r'(x) - r(x))$$

Proof. If we can prove that the amortized cost is $3(r'(x) - r(x))$ for a double rotation, then we can simply telescope our sum to prove the key lemma. So, we will focus on a single double rotation. The $+1$ in the lemma itself comes from the potential single rotation we may need to do at the end.

When we do a zig-zig rotation, the real cost is 2 rotations. The potential of all the nodes not being rotated doesn't change. Thus, the only potential changes we have to consider are that of x, y, z. The change in potential is $\Delta\Phi = (r'(x) - r(x)) + (r'(y) - r(y)) + (r'(z) - r(z))$. The rank of x increases while the rank of z decreases, and the potential of y can either increase or decrease.

Intuitively, if $r'(x) \gg r(x)$, then the cost of rotation is easily paid. If $r'(x) \approx r(x)$, then the cost of rotation is not paid with x's rotation, but we do have the additional information that subtree A is pretty fat, which means that the decrease in y and z potentials can be used to pay for the rotation.

To formalize this, we know that $r'(x) = r(z)$ by definition, and likewise $r'(y) < r'(x)$ and $r(y) < r(x)$. The actual cost is as follows:

$$\text{cost} = 2 + \Delta\Phi \leq 2(r'(z) - r(x)) + (r'(x) - r(x)).$$

We will prove our lemma then if

$$2 + (r'(z) - r(x)) \leq 2(r'(x) - r(x)) \iff \log_2\left(\frac{s'(z)}{s'(x)}\right) + \log_2\left(\frac{s(x)}{s'(x)}\right) \leq -2$$

Letting |S| denote $s(S)$, looking at the effect of rotation, we have that

$$s'(z) + s(x) = |A| + |B| + |C| + |D| < s'(x) = |A| + |B| + |C| + |D| + 2.$$

Since the maximum value of the function $\log_2(x) + \log_2(1-x)$ is -2, after substitution, we have that

$$\log_2\left(\frac{s'(z)}{s'(x)}\right) + \log_2\left(\frac{s(x)}{s'(x)}\right) \leq \log_2\left(\frac{|C|+|D|}{|A|+|B|+|C|+|D|}\right) + \log_2\left(\frac{|A|+|B|}{|A|+|B|+|C|+|D|}\right) \leq -2$$

which proves the Access Lemma for a zig-zig rotation.

The zig-zag case is left as an exercise. □

With this lemma, we can prove our desired result:

Corollary 4.2 (Balance Theorem)
A splay operation takes amortized time $O(\log n)$

Proof. If all the weights are equal to 1, we know that $r'(x) = \log n$. By the Access Lemma, the splay time is then $O(\log n)$. Since the splay time is doing much more work than a find operation, then the entire search operation is $O(\log n)$. □

4.4 Results with the Access Lemma

When the weights are not all equal to 1, we can try to bound the potential in order to derive more results about the splay tree. The actual cost is the amortized cost minus the change in potential, which supposing we have m operations, is $O(m \log n) - \Delta\Phi$. How large is this change in potential?

We can derive a few simple bounds. Letting the weights be w_x and having total sum W, we have that $\Phi(0) \leq n \log W$, which corresponds to the case where everything is a root. Likewise, we can also get a nice lower bound of $\Phi(m) \geq \sum \log w_x$, where everything is a leaf. Thus, $\Phi(0) - \Phi(m)$, the extra amortized cost, is $\sum \log \frac{W}{w_x}$. Notice that the terms in the summation $\log \frac{W}{w_x}$ are upper bounded by the time it takes to splay the node x! This tells us that the change in potential for the amortized cost is negligible and the real cost of splaying is approximately equal to the amortized cost.

With the access lemma, we can prove many results, for example, the static optimality theorem:

Example 4.3

Suppose we perform m operations, such that item x is accessed with probability p_x. To optimize the time of performing the operations, the best thing to do is to put the most-accessed items near the root. Since each level k has 2^k spots, any item with $p_x \geq 2^{-k}$ can go to level k. Thus, the overall search cost is $m \sum -p_x \log_2 p_x = mS$, where $S = -\sum p_x \log_2 p_x$ is the **entropy**.

Theorem 4.4 (Static Optimality Theorem)

Splay trees achieve this search cost as well, without knowing the individual p_x.

Proof. Let the weights $w_x = p_x$. By the access lemma, the cost of each element access is $\log_2 \frac{W}{w_x} = \log \frac{1}{p_x}$. The conclusion follows. □

We will define a **static finger**, which comes from the idea of locality of reference. The idea is that nearby data will most likely be accessed, and so we may start a search from a 'finger' rather than a root. For certain types of accesses, starting from a 'finger' rather than the root is much faster, and splay trees do just as well. Additionally, even if the finger is dynamic and can move around, the splay tree also matches the time complexity.

What can't we do with splay trees? Well, we think that they're actually always optimal - there is a long standing unresolved conjecture about them:

Conjecture 4.5 (Dynamic Optimality Conjecture)

Define a **binary search tree strategy** to be one in which we can do whatever operations we want in a binary tree, with complete knowledge of node requests. No matter what binary search tree strategy we have, splay trees will match the performance. No counterexamples have been found so far.

Joke 4.6. *This conjecture will be a homework problem; anyone who solves it gets an A.*

Remark 4.7. *The current state of the art towards the conjecture is from Demaine (another MIT prof), who introduced Tango Trees, which were provably shown to perform within $O(\log \log n)$ of the optimal binary search tree strategy.*

5 Lecture 5: Splay Updates, Buckets

5.1 Wrapping up Splay Trees

Recall that previously we showed that the find operation in splay trees was amortized $O(\log n)$ time and we also got other bounds by changing the weights in our potential function. While we could just use a standard insert/delete operation, this is unacceptable, since without a splay a linked-list-like splay tree will have $O(n)$ insertion. So, we can just extend this easily - for an insertion, we first insert the element, then splay, for an amortized runtime of $O(\log n)$. However, since we inserted a new item, this also leads to an increase in potential and our analysis has to be made much more complicated. The analysis for delete is even worse, since we need to choose what exactly to splay.

Let's try alternatives. Specifically, we'll define the **split** and **join** operations as follows:

Definition 5.1

A **split** operation takes in a node as an argument, and returns two trees - one of which contains the items with key $\leq x$, and the other containing the items with key $> x$.

A converse operation, **join**, will take in two trees T, S such that T has all items with key less than any element in S, and return their union.

To implement these two operations, it is much simpler than our previous idea. For a split, we simply splay x (or its predecessor if x doesn't exist), and then remove the right subtree, creating two subtrees that satisfy the requisite conditions. The amortized time is $O(\log n)$, and the potential obviously. For a join, we simply splay the minimum element of tree S to the root, and then joins tree T as a child of the minimum element of S. The potential increases by $O(\log n)$ which is fine, so the amortized cost is still $O(\log n)$.

Now that we have the split and join operations defined, we will implement insertion by simply calling split(T, x), giving us subtrees A and B with $A < B$, then calling join$(A$, join$(x, B))$. For deletion, we can first split(T,x), creating subtrees $A \to x$ and B. Removing the element x and then joining A and B effects deletion. Both take time $O(\log n)$

When we implement splay trees in practice, we can use top-down splaying and to splay only on a long path, to improve runtime by a constant factor. The decrease in time is actually due to memory structures itself, since reading memory is much faster than writing memory. Another heuristic is to stop splaying after a while, which works since splay trees keep frequently accessed elements on the top. This heuristic is dangerous, however, if the probability distribution of inputs change.

Note 5.2

Another usage of splay trees is actually as a compression algorithm. Suppose that you want to compress a sequence of characters. We store them as a splay tree, but instead of sending a character over the communication link, we send the path in the tree, and then splay the element afterwards. The number of bits sent over the communication is equal to the search time, and so this is a way to show an information-theoretic result with data structures.

5.2 Buckets and Indirect Addressing

Previously, the things we have done were all with pointer-based structures, and we could essentially replace any arrays that we had with a pointer-based structure. But there are actually applications where arrays are optimal, for example, shortest paths when the edge weights are integers in $[1, C]$. Recall that the shortest path problem can be solved in time $O(m + n \log n)$ with Fibonacci heaps, or $O(m \log n)$ with standard methods. If $C = 1$, then a BFS works an just takes $O(m)$ time. To generalize, we can split up a weight w into just a chain of w edges, and this takes time $O(mC)$, which beats the previous algorithm for small C.

Let's go back to our ideas about priority queues, which we used in Dijkstra's algorithm. One thing to note is that in the priority queue, the minimum distance is nondecreasing. If we can make a specialized priority queue for this case, then we may be able to simplify.

Dial said to simply make an array of buckets to do so. In bucket d, we put all the items of distance estimate d. For example, if we start with source s, then we just insert the neighbors in their requisite buckets, and then remove the source from the array. Afterwards, as in Dijkstra's, we need to find the new min. This we can do simply by going to the next bucket, via a forward scan! We need to do a forward scan here, since there are too many overlapping min neighbors from deleting the other elements, and it is hard to keep track of the min auxillarily.

Our algorithm will work as follows: When we have a node in the priority queue, then we will do a forward scan to the minimum, remove the minimum, and then insert the minimum's neighbors in our bucket. We repeat this, with our forward scan starting from the previous minimum.

The time it takes for neighbor updates is $O(m)$, and the time for scans should just require $O(\text{max distance}) = O(D)$. D itself is bounded by nC, and so the overall runtime is $O(m + nC)$ which is better than before!

The space we require is $O(n + D) = O(nC)$. But we can do better. The range of relevant values within the array is from d to $d + C$. So, if we store our array in a circle mod $C + 1$, by using the same index to hold different values, none of the values will be occupied at the same time. Thus, with a circular array, we only need $O(n+C)$ space, which is optimal.

Joke 5.3. *This algorithm, Dial's Algorithm, was invented in '73, when people could get things named after themselves with only arrays. Now, we need fancy data structures to get something named after ourselves.*

5.2.1 Improvements

This is an algorithm that cannot be beaten with a fancy data structure, since the constants are good. However, if C is large, then there is possibility for improvement. The largest time cost that we have is with the forward scan, and we can optimize this. We'll do so with a **2-level bucket scheme**, where we auxillarily make blocks of size b that records how many buckets in each block which are nonempty, that are updated when elements are deleted/added. Now, on a scan, we can simply traverse blocks until we get to a nonempty bucket, and then traverse the nonempty bucket.

What are the improvements of runtime? The neighbor updates are still $O(m)$. The bucket scan is $O(b)$ per delete-min, for a total time of $O(nb)$. The block scan takes time $O(nC/b)$, since we go to at most bucket nC and can skip b buckets every time. Choosing $b=\sqrt{C}$ causes the total time to be reduced to $O(m+n\sqrt{C})$

But if we can do it once, we can do it multiple times. Let's try a **3-level bucket scheme**, where we have superblocks that store blocks, which store buckets. Then, we scan for a non-empty superblock, then scan it for a non-empty block, then scan that for a nonempty bucket. The optimal time is from setting the block size to be $C^{1/3}$ and the superblocks size $C^{2/3}$, for a total runtime of $O(m+nC^{1/3})$

When we have a **k-level bucket scheme**, then we can make the exponent of C very small. But this is a problem, since when we have more and more layers, we need $O(k)$ time to update from all our insertions, and also need to scan k layers. The overall runtime becomes $O(k(m+n\cdot C^{1/k}))$.

5.2.2 Formalization with Tries

This idea can be formalized to a **trie**, which is a depth k tree over sibling arrays of size Δ. Essentially, every element within the trie is an array of size Δ, which itself also contains a subtrie of depth $k-1$. We have a range of C possible values (once again, thinking in a circle). Ideally, we choose $C+1=\Delta^k$ to store all the elements under consideration, and hopefully Δ is a power of two (we'll see why later).

Let's consider finding minima in this trie. Insertion and Deletion is $O(k)$ per operation, since that's just the time needed to traverse down the trie. For the delete-min operation, we need to first find the minimum nonempty child of root, which takes time Δ. Then, we can go down to look at the array pointed to by that element, and so on, until we reach the bottom layer of the trie. This takes time $O(k\Delta)$. The overall runtime is then $O(mk + nk\Delta)$, which when minimized for k, becomes $O(m \log_{m/n} C)$. This is linear for a dense graph and equal to $O(m \log C)$ for a sparse graph.

5.2.3 Laziness wins again

How do we make this even better? Be lazy! Denardo and Fox ('79) proposed to modify the insert operation to procrastinate more and be more lazy: "Don't push items down until we must." We'll only make a child in the trie if we have more than one element in the node itself, essentially only expanding as necessary. This is a good thing to do, but if we're only doing inserts and deletes, we don't even need to compare things for inserts and deletes! Therefore, we will only invest time into expanding a node into its child array if we need to scan that bucket when we do a delete-min. We will then only have one "active" trie node per level, which are on the path to the current min. For the other elements, we just let them stay in the bucket in the lowest active node they belong to. To quickly check emptiness, we will keep track of the number of items in each layer (not descendants).

In implementation, for insert, we start at the top of the trie. We walk down until we land in a non-minimum bucket, and increment that level's item count, and then stop. For the decrease-key operation, we may remove it from the current bucket, and find its new bucket, which is either behind or below its current bucket. For implementation, we simply check to see if our element can fit into the nearest previous bucket, and if they do, we may need to go down another layer. Since the min is monotonic, then we have no risk of going beyond our expanded bucket structure. The delete-min operation can be implemented by actually removing the min, and then checking the number of items in each layer. If the layer is non-empty, then we just do a forward scan. If the layer is empty, then we go up a layer, and do a forward scan to find the bucket in the layer above, and then expanding the bucket of the new min. Overall, we keep going up until we get to a non-empty layer, forward scan for the first non-empty bucket, and then expand it to the bottom.

The real cost for insert or decrease-key is $O(k)$, since we simply traverse down the trie. If we amortize the decrease-key cost into the insertion, then decrease-keys become $O(1)$. For a delete-min, we need $O(\Delta)$ to scan, then $O(k)$ to go up and down the trie, for a total time of $O(k + \Delta)$.

Thus, with laziness, the runtime is $O(m + n(k + \Delta))$, which when balanced (here a power of 2 for Δ is nice), becomes $O\left(m + \frac{n \log C}{\log \log C}\right)$. It's not a big improvement, but it is from simple data structure, and $\log \log$ factors are relatively important in data structures. This can be improved even further with a priority queue on the scan, which we'll implement in the next lecture.

6 Lecture 6: VEB queues, Hashing

6.1 Improvements to the Denardo and Fox queue

When we do a delete-min in our Denardo and Fox queue from previously, we have to do a forward scan to find the smallest non-empty bucket. But this may take a decent time, while it is just a problem in finding the next smaller number, a perfect application for a priority queue! So, Cherkassky, Goldberg, and Silverstein capitalized on this insight by making a Denardo and Fox queue, plus adding a standard heap to find the min entry at each level. (This is known as a **HoT queue**, heap on top queue.) Essentially, the top layer of the trie is made much bigger and into a heap. If we make the size of the heap 2^{Δ}, the cost of heap traversal is negligible compared to the rest of the structure, but the advantage is that we can actually reduce the number of layers needed to store values. This ultimately changes the runtime to $O\left(m + n(\log C)^{1/3}\right)$.

Remark 6.1. *Goldberg, one of the inventors of this improvement, is a big practitioner of Experimental Algorithms. Expermientalists actually implement the algorithm and then see how it can be improved - what heuristics can improve access time, etc? These can also lead to new theorems, and Goldberg's papers are a great demonstration of these techniques.*

Remark 6.2. *Silverstein was Prof. Karger's brother's roommate and also Prof. Karger's classmate in graduate school. Silverstein dropped out of graduate school to work at Google, and now is doing great work at Khan Academy, showing how a deep study of algorithms can lead to many places.*

6.2 VEB queues

This insight in the HoT queue actually comes from a previously introduced data structure, **van Emde Boas trees** (1973). The general idea of VEB trees is to implement a priority queue as a two-layer trie, with each layer having $\sqrt{C}$ elements. We will work with b-bit words ($C = 2^b$).

The structure of the queue Q is as follows:

- $Q.min$: This is stored in Q, but not in the tree itself (this is critical!)
- $Q.low$: This is an array of size $\sqrt{C}$. The intuition is that each will have $b/2$ bits. Each of the values in the array point to another VEB queue on $\sqrt{C}$ values in range.
- $Q.high$: This is another VEB queue, which stores the indexes of nonempty level-2 blocks. The maximum size of $Q.high$ is $\sqrt{C}$, each with $b/2$ bits.

This gives us a priority queue in $Q.high$, which allows us to easily access the minimum elements, as wanted. Also, note that the number of layers we have is $\log b = \log\log C$, a significant improvement.

Let's now implement our operations as follows:

- For our insertion, if $x < Q.min$, we will just swap x and $Q.min$, and the problem reduces to just inserting an element greater than the min. Now, we will say that $x = 2^{b/2}x_h + x_l$. We check if the queue $Q.low[x_h]$ is nonempty. If it is nonempty, we simply insert x_l into $Q.low[x_h]$. Otherwise, we just make a VEB queue there, which we can insert x_l into, making sure to also insert x_h into $Q.high$ to keep track of the new queue.
- For delete-min, we can simply query $Q.min$. We remove it, and then we need to replace it with the minimum of the recursive structure. To do this, we look at $Q.high.min$, the index of the first nonempty level-2 block (or equivalently, the high bits of the new min). A special case occurs if $Q.high.min = null$, this means that the recursive structure is empty, and so there are no more elements in the queue. Otherwise, $Q.high.min$ tells us x_h, and deleting the min from $Q.low[x_h]$ gives us x_l. Finally, if deleting the min from the second layer results in an empty queue, we need to delete it from $Q.high$ in order to keep our queue consistent. The new minimum is $2^{b/2}x_h + x_l$.

Let's do a quick runtime analysis. Insertion of a b bit integer is $T(b) = 1 + 2T(b/2)$, which occurs when $Q.low[x_h]$ is not made yet, needing to insert two integers on size $b/2$ queues. Using Master's Theorem tells us that insertion is $O(b)$. However, we can improve on this recurrence formula, since insertion x_l of into an empty queue (happening if $Q.low[x_h]$ is previously empty) only takes time $O(1)$! Our recurrence then becomes $T(b) = 1 + T(b/2) = O(\log b) = O(\log \log C)$.

Likewise, for a deletion, the recurrence also looks like $T(b) = 1 + 2T(b/2)$, but the deletion of the level-2 min is just $O(1)$, and so the actual recurrence is $T(b) = 1 + T(b/2) = O(\log \log C)$. Note that here, if $Q.min$ was not stored separately, we could not do the simplification of the recurrence.

VEBs are exponentially better than every other heap structure we've seen. But it has a drawback in the space required. For a b bit queue, we need approximately $2^{b/2}$ space for the top layer, and then another $2^{b/2}$ space for every item we insert.

Can we do better? Well, the arrays themselves are only used for guaranteeing $O(1)$ lookup. If we can find another data structure to do $O(1)$ lookup with significantly better lookup efficiency (spoiler alert: hash tables), then the space complexity becomes much better.

6.3 Hashing

Note 6.3

Hashing is an important topic to understand; one reason is since it involves randomization, which was a stated prerequisite of the course. We'll use hashing to create dictionaries, which are data structures that store key/value pairs, with insertion, deletion, and lookup key operations. While hashing should have been covered in standard undergraduate algorithms courses, we will now prove important results without the assumptions made in those previous courses.

For our model, we will make our keys integers in $[m] = \{1, 2, \ldots, m\}$, which can be stored in an array of size m with $O(1)$ lookup. But we can do better. Supposing that we only have n keys to insert, the question is to ask if we can use $s > n$ space to store everything in $O(1)$.

To do so, we'll use a **hash function** $h : [m] \to [s]$, and to store each key k in an array at position $h(k)$. The problem with a standard hash function is with collision of values $(h(k_1) = h(k_2))$. We can get around it by storing all the values that hash to one particular value in a linked list, but then this makes the access time potentially not $O(1)$.

We need a good hash function, one that causes few collisions. However, this is impossible:

Theorem 6.4

There are no good hash functions.

Proof. This is impossible due to the pigeonhole principle, which says that at least one bucket will have at least $\frac{m}{s}$ elements. If we choose the input keys devilishly, the lookup time for that bucket is $O(m/s) = o(1)$, which means that that hash function is not good. □

To get around this, we'll instead use randomization, with a **hash family**, which is simply a set of hash functions such that we can pick a good one for any set of items.

What if we just pick a random function from our hash family?

Theorem 6.5

If n keys randomly distributed in $[s]$ are hashed, then the expected access time in our hashing function is $O(1 + \frac{n}{s})$.

Proof. We will define the indicator random variable C_{ij} as follows: $C_{ij} = \begin{cases} 1 & \text{if items } i, j \text{ collide} \\ 0 & \text{otherwise} \end{cases}$.

The time to find element i is simply $1 + \sum_j C_{ij}$. The expected time to find an element is then $\mathbb{E}[1 + \sum_j C_{ij}] = 1 + \sum \mathbb{E}[C_{ij}] = 1 + \sum \mathbb{P}[\text{collision between } i \text{ and } j]$ by linearity of expectation.

The probability that two randomly assigned elements $i, j \in [s]$ collide is simply $\frac{1}{s}$. Then, the expectation value of required time simply becomes $1 + \frac{n}{s}$, as desired. $\square$

Even though we've proven that nice theorem, this doesn't help us, since remembering all these random functions take a huge amount of space! Specifically, there are m^s functions from $[m] \to [s]$, taking up $m \log s$ space (this bound comes from information theory).

Getting around this, Carter and Wegman introduced **2-universal hash families**. This comes from the observation that we don't need a completely random hash function, but rather just a function that is **pairwise independent**, such that any two elements in $[s]$ will have a probability $\frac{1}{s}$ of collision. Thus, if we can find a way to generate pairwise independent hash functions with less space, then we get around the bottleneck that our previous attempt at random hash functions had. While pairwise independence sounds like it would trivially lead to **mutual independence** (independence of all elements), the following example demonstrates that this assumption is false:

Example 6.6

Let's consider flipping three coins x, y, z. The probability that any pair of two coin flips are the same is just $\frac{1}{2}$.

What if we're lazy? Let's just flip two coins x, y, and set $z = x \oplus y$ (exclusive or). The probability that any pair of two coin flips are the same is still $\frac{1}{2}$! But these values are obviously not independent.

This example shows that with just two bits, we can make three pairwise independent variables, which provides an avenue to produce the space needed for our hash function storage.

7 Lecture 7: Hashing, Max Flows

7.1 Hashing with Less Space

Let's try to come up with a way to generate pairwise random independent functions. We'll suppose that the hash table size is a prime number p. The **Carter-Wegman assumption** was to map each key according to $h_{ab} : x \mapsto ax + b \bmod p$.

Theorem 7.1

If a, b are uniformly randomly chosen from $0, \ldots, p-1$, then for fixed $x \neq y$, $h_{ab}(x)$ and $h_{ab}(y)$ are uniform and pairwise independent over $0, \ldots, p-1$.

Proof. To show this claim, we need to prove that $\mathbb{P}[h_{ab}(x) = s \text{ and } h_{ab}(x) = t] = \frac{1}{p^2}$ for any fixed $s, t \in \{0, \ldots, p-1\}$.

Working in $\mathbb{F}_p$, this event happens if and only if $ax + b = s$ and $ay + b = t$, which when expressed in matrix notation is just when

$$\begin{pmatrix} x & 1 \\ y & 1 \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} s \\ t \end{pmatrix}$$

Since the determinant of the square matrix is $x - y \neq 0$, this means that the inverse of the square matrix exists, and hence there is only one pair of (a, b) that will lead to our desired event happening. There are p^2 pairs of (a, b), and since our (a, b) are random, we are done. $\square$

This means that if our hash table is prime-sized, we can get $O(1)$ expected lookups, even when we choose $n = s$. What about if the size of the hash table is non prime-sized?

Theorem 7.2

$O(1)$ lookups even if the hash table size s is not prime are possible, when we choose $p \gg s$ and assuming that $x \in \mathbb{F}_p$. The hash functions are defined as $h_{ab} : x \mapsto ((ax + b) \bmod p) \bmod s$.

Proof. We know from previously that the function $(ax + b) \bmod p$ results in a uniform distribution in the range $0, \ldots, p-1$. Then, when we take mod s, out of the p total values in $\mathbb{F}_p$, the number of values that map to a certain residue mod s is either $\left\lfloor \frac{p}{s} \right\rfloor$ or $\left\lfloor \frac{p}{s} \right\rfloor + 1$. The maximum difference of residue probabilities is then $\frac{1}{p}$, which we can make arbitrarily small by choosing p to be large (for example, letting $p = n$.) This minor perturbation in probabilities still guarantees us $O(1)$ lookups. $\square$

Remark 7.3. *Usually in cases like these, p could also be a prime power and everything could still work out, for example, $p = 2^{64}$.*

We've now shown that $O(1)$ expected lookup is achievable. What about the **maximum load**, the most items that any one bucket holds in a hash table?

Claim 7.4
The maximum load is $O(\sqrt{n})$ with probability $1 - \frac{1}{n}$, when $n = s$.

Proof. Exercise. □

Some items will probably have a larger load, but this shouldn't affect runtime, since it is unlikely that a certain key is accessed many times.

Remark 7.5. *There are pairwise independent hash families where having $O(\sqrt{n})$ max load is very likely, but this doesn't affect the average load at all. So no worries!*

7.2 Perfect Hash Functions

Question 7.6. *Can we guarantee worst-case $O(1)$ lookups, with no collisions, if we are given the keys in advance? (this is known as a perfect hash function)*

Well, let's first check the chance of no collisions happening if a random hash function is used. We'll try to calculate the expected total number of pairs of collisions:

$$\mathbb{E}\left(\sum_{i<j} C_{ij}\right) = \sum_{i<j} \mathbb{E}(C_{ij}) = \sum_{i<j} \frac{1}{s} = \binom{n}{2}\frac{1}{s} \approx \frac{n^2}{2s}$$

If $s \geq n^2$, then the expected number of pairs of collisions happening will be $\frac{1}{2}$. How do we convert this expectation to a probability? The answer is Markov's Inequality:

Lemma 7.7 (Markov's Inequality)
If $X \geq 0$ is a random variable, then $\mathbb{P}[X \geq t] \leq \frac{\mathbb{E}[X]}{t}$.

With Markov's Inequality, we get that the probability of a collision happening is less than $\frac{1}{2}$. This probability of $\frac{1}{2}$ is far from guarantee, but we can simply try again if we fail. After n tries of a perfect hash function, the probability we succeed is $1 - \frac{1}{2^n}$.

How long does this generation take? We need to take into account how many times we need to try to find the perfect hash function, as well as the time it takes to generate and test a perfect hash function. The time it takes to test the perfect hash function is $O(n)$ and the expected number of times we go through the loop is just 2, so the expected time for hash function generation is $O(n)$.

Note 7.8

The procedure above to generate one hash function is a **Monte-Carlo algorithm**, which is a fast algorithm that is likely to succeed. A **Las Vegas algorithm** is one which is often fast and always works, which describes our overall perfect hash function generation. We can turn any Monte-Carlo algorithm into a Las Vegas algorithm similarly to above, just repeating until we succeed. We can also go the other way, though this is a bit harder to describe.

This 'perfect hash function,' though, is not perfect at all, since we are using n^2 space for n items. Is it possible that our $\frac{n^2}{2s}$ upper bound on the number of collisions is not tight? Let's demonstrate that this is false with a quick example:

Example 7.9 (Birthday Paradox)

We will hash every person into 365 buckets according to their birthday. When we have ≥ 23 people, the probability of collision is $\frac{1}{2}$. When we have 37 people (the class size today), it is almost certain that we have a collision. It turns out that the probability of no collision happening is extraordinarily small, and our $\frac{n^2}{2s}$ bound is actually pretty tight.

To show this, we'll check the probability that we have no collision is

$$\left(1-\frac{1}{s}\right)\left(1-\frac{2}{s}\right)\cdots\left(1-\frac{n-1}{s}\right) \leq e^{-\frac{1}{s}-\frac{2}{s}-\dots-\frac{n-1}{s}} \leq e^{-n^2/s}$$

and so if we don't make our hash table quadratic in size, we are almost certain to see collisions.

Joke 7.10. *As MIT students, we can abuse this paradox and bet that there exists two people with the same birthday in a bar, almost certainly getting a free drink.*

We get around this problem with a two-level hashing idea introduced by Fredman, Komlos, and Szemered, who are all well-known algorithmists. First, we'll try to hash our n items into $O(n)$ space. Consider the previously mentioned hashing with chaining method, where we have a linked list to resolve collisions. But why do we even use a linked list? Why not replace it with a perfect hash table on each bucket?

Definition 7.11

For our analysis and convenience, we will define the **Iverson Bracket** of a logical statement to be equal to 1 if the statement is true and 0 otherwise. For example, $[x > 0] = 1$ if $x > 0$.

Theorem 7.12

Supposing that we make k secondary hash tables with size b_k, this hashing scheme takes $O(n)$ expected space and maintains $O(1)$ lookup.

Proof. For lookup, simply go to the perfect hash table specified by the first hash function, and then find the element in $O(1)$ time. The space usage for the secondary buckets is

$$\sum_k b_k^2 = \sum_k \left(\sum_i [i \in b_k]\right)^2 = \sum_k \left(\sum_{i,j} [i \in b_k][j \in b_k]\right) = \sum_{i,j} \left(\sum_k [i \in b_k][j \in b_k]\right) = \sum_{i,j} C_{ij}$$

The expected space usage is then

$$\mathbb{E}\left(n + \sum_{i,j} C_{ij}\right) = n + \mathbb{E}\left(\sum_i C_{ii}\right) + 2 \cdot \mathbb{E}\left(\sum_{i,j} C_{ij}\right) = n + n + \frac{n^2}{s} = O(n).$$

□

Corollary 7.13

This hashing scheme can be modified to take $O(n)$ *worst-case* space.

Proof. We'll take a similar approach as we did with regard to the quadratic perfect hash function itself. We will generate a perfect hash table according to above, and check its space usage. If it is greater than two times the expected, then we repeat our procedure. A simple application of Markov's Inequality then guarantees that we can generate a $O(n)$ space perfect hash table. □

Note 7.14

The expected space usage is approximately $13n$ if we proceed as indicated above. With a few optimizations, this space is cut to approximately $6n$. In the original paper, however, this space was cut to $(1 + o(1))n$ space, a truly remarkable achievement. This is achieved by using the empty buckets in the primary hash table to hold information about the secondary hash functions, something that should "never be done in practice," but is fine from a theoretical perspective.

This method of perfect hashing required us to check whether everything is perfect or not. But this is impossible if we don't know the keys - for example, if we want to support insertion or deletion. This isn't actually too hard, and will probably be on the PSET next week. The method works by just recreating the hash table when there are too many imperfect things, and all we need to show is that we don't need to do this too often.

7.3 Combinatorical Optimization

We've now wrapped up the data-structure part of the course, and now we're going to move onto combinatorical optimization. A **combinatorical optimization problem** is one in which we have a set of feasible solutions, in which each has a cost or a value. Our goal is to maximize the value or minimize the cost of the solution, and this is our output. When we work with such problems, we're now not going to care about runtime of an operation. We just care about the runtime of making the correct output.

Our general workflow for tackling combinatorical optimization problems are as follows:

1. First, understand feasibility - what does it mean when a solution is actually feasible?
2. Then, develop an algorithm to determine if a solution is feasible or not.
3. Afterwards, with our algorithm, we can find a certain feasible solution, showing that they exist.
4. We then develop an algorithm to verify the optimality of a feasible solution.
5. Finally, we compute our optimal solution.

7.4 Introduction to the Max Flow Problem

Definition 7.15

The **max flow problem** involves a directed graph G with m edges and n vertices, where we have a source vertex s and a sink vertex t. Each edge e has a capacity $u_e \geq 0$. We will define a **flow** f_e, a number per edge, and we will call the flow feasible if $0 \leq f_e \leq u_e$.

Additionally, all vertices $v \neq s, t$ need to satisfy the conservation constraint, which states that $\sum_w f(v, w) - f(w, v) = 0$. In other words, whatever flow comes in must go out. We define the value of a flow to be $\sum_v f(s, v) - f(v, s)$, which is the amount of flow leaving the source. Our problem is in finding the max value of the flow.

This problem is known as the 'central problem' in combinatorical optimization, and was first introduced by Ford and Fulkerson in 1956, due to its wide variety of solutions and applications. Many problems can be reduced to a max flow problem. Currently, our best algorithms can run in almost linear time.

8 Lecture 8: Max Flows I

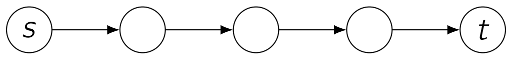
As we mentioned previously, to solve this combinatorical optimization problem of max flow, we first need to feasibility conditions. While earlier we defined the 'gross flow' per vertex, there is also another definition of flow based on edges, the 'net flow.'

Definition 8.1
We define the **net flow** on an edge e connecting nodes u, v to be $g(v, w) = f(v, w) - f(w, v)$, where $f(v, w)$ is the amount of flow from v to w.

This net flow will have certain properties:

- $g(v, w) = -g(v, w)$ (skew-symmetry)
- $\sum g(v, w) = 0$ (conservation)
- $g(u, v) \leq u_e$.

Our simplest feasible flow is just the 0 flow, which is generally always feasible. The next simplest flow is just a straight path from the source to the sink, where each edge has the same flow that is less than the capacities of each of the edges:



In general, we can have many separate flows, which can be broken down into what we call as $s - t$ paths and cycles, which are respectively flow paths that go from the source to the sink and paths which go in a cycle.

Lemma 8.2 (Path Decomposition Lemma)
In fact, these are the only types of basic flows. Any $s - t$ flow can be decomposed as a sum of $s - t$ paths and cycles.

Proof. We will induct on the number of edges carrying nonzero flow. Given a nonzero flow, our plan will be to find a nonzero path, and then remove it from the flow, which then will show the Path Decomposition Lemma through (backwards) induction.

We start at the source node s. Since the flow is nonzero, the edge leaving s is carrying flow. Let's say that the flow goes to some vertex v. The edge leaving v is also carrying flow by conservation, and we can continue along this edge until we reach t or we reach a cycle (this has to happen since there are only finitely many nodes, and only the flow that can 'disappear' is at t).

In either case, we've either found a valid $s - t$ path or a valid cycle. Now, we will look at the min flow edge, and decrease the flow on each edge on this path/cycle by that much.

We finally just have to verify that what remains is still a flow. Clearly the capacity constraints still hold. Since we're not changing the net flow between each vertex (except at the source and the sink), the conservation requirements still hold.

Since we zero the flow on the min edge in each iteration, this process terminates. In fact, this argument also proves that we can decompose a flow into $\leq m$ paths and cycles. $\square$

Now that we've shown some conditions on feasible flows, our next question to ask is a decision problem: Is there a nonzero flow in G?

Theorem 8.3

A nonzero flow only exists if and only if t is reachable from s through edges of nonzero capacity.

Proof. If t is reachable from s, then this means that there is a $s - t$ path. We can simply follow the flow along this $s - t$ path (guaranteed to have nonzero flow) to show that a nonzero flow exists on the graph.

For the other direction, we use the flow decomposition lemma. If we have any path from s to t, then t is obviously reachable from s. Otherwise, if we only have cycles, we know that any cycle through s gives net flow zero. This means that the total flow is 0. Thus, the only time a nonzero flow exists is when there is a $s - t$ path. $\square$

Let's turn this question around now - suppose that we have no feasible flow. How do we show so, to someone else? (note: this is called a **certificate**, which is some data that allows for easy checking of feasibility).

To do so, we'll partition the graph vertices into two sets, where the set S of vertices are all those that can be reached from the source, and the set $\overline{S}$ is the complement of S. We call in general any partition a **cut** of the graph. In this case, there is no edge connecting S and $\overline{S}$,and this cut $(S, \overline{S})$ can act as a certificate for showing that no flow exists.

Now let's change these ideas a bit to verify optimality in the max flow problem. We can find an upper bound on the flow, and then show that the flow is tight.

Theorem 8.4

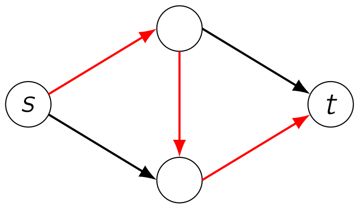
Suppose that we have a partition $(S, \overline{S})$. Then, the max flow is less than $\sum_{e \text{ crossing } S \to \overline{S}} u_e$.

Proof. We'll decompose our flow into $s-t$ paths. Each of the $s-t$ cuts crosses the $S, \overline{S}$ cut, and can send their maximum capacity across the $(S, \overline{S})$ partition. The total amount of flow that can be sent is then the sum of these values. $\square$

Defining the **capacity** of the cut as $\sum_{e \text{ crossing } S \to \overline{S}} u_e$, we've show that the max flow is $\leq$ the capacity of any cut. Thus, the minimum cut is an upper bound on the max flow. We don't know how good this bound is (spoiler alert: it's tight), but it is a upper bound.

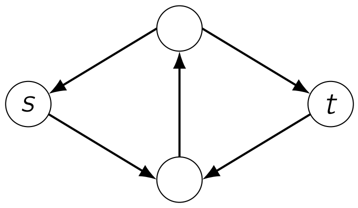
Now, suppose we have a flow which is not maximum. How do we make it better?

Let's consider our example graph as follows, with a example flow highlighted as below, and with capacities all equal to one:



Obviously, the flow is not maximum here - the maximum flow is equal to 2. We can try to make it better by adding flow on an available path, but in this case, we can't, since there are no more available paths. Instead, we need to be able to reverse some paths.

To formalize, given a flow f, we'll define the **residual capacity** $u_f(v, w) = u(v, w) - f(v, w)$. We also have that $u_f(w, v) = u(w, v) + f(v, w)$. We define the **residual graph** to be one in which the capacity is equal to the residual capacity. For our example, our residual graph is as follows:



With our residual graph, we have an easy way to add more flow - simply search for an **augmenting path**, a $s-t$ path with nonzero flow in the residual graph. Then, we can simply add this $s-t$ path to the original flow, to generate a flow with higher value.

What if there is no augmenting path in the residual graph? Then we have a max flow, as guaranteed by the central Max-Flow Min-Cut Theorem:

Theorem 8.5 (Max-Flow Min-Cut Theorem)
The following are equivalent:

1. f is the max flow.
2. There are no augmenting paths in G_f.
3. The max flow is equal to the capacity of a certain cut $(S, \overline{S})$, which is the minimum cut.

Proof. If there is an augmenting path, then we can increase the max flow, and hence the flow is not maximum. Thus, if the flow is maximum, then there cannot be any augmenting paths. This shows (1) implies (2).

If there are no augmenting paths, let S be the set of reachable vertices in the residual graph. All the edges leaving S in the original graph are **saturated** (i.e. are sending a max capacity flow) and those entering S are empty, due to our definition of residual graph.

Now, let us consider a path decomposition of the original graph. Each $s - t$ path crosses the cut exactly once, since the edges entering S are empty and thus flow cannot go backwards. The capacity of this cut then equals the flow of the graph. Further, since the capacity of a cut is an upper bound on the flow, this means that this particular cut is actually the minimum possible cut. This means that (2) implies (3).

Finally, (3) implies (1): The capacity of a cut is an upper bound on the flow, and if we achieve equality, then this means that the given flow must be the maximum possible. □

From this theorem, we see that min cuts and max flows are intrinsically linked, and thus one can act as a certificate for the other. Additionally, we have shown the following:

Corollary 8.6
The net flow across any cut is equal to the flow value.

Remark 8.7. *The total number of max flows and min cuts are not correlated. It is possible to build graphs with multiple maximum flows and a unique min cut, and vice versa.*

8.1 Max Flow Algorithms

We've actually already went over an algorithm for finding the max flow. We look at the residual graph, and find an augmenting path. We then augment our flow, increasing the flow by the minimum residual capacity along this path, and repeat until there are no more augmenting paths.

Let's now look at runtime. We can find any path in G_f in time $O(m)$ by breadth-first search. How many iterations do we need? If we have integer capacities, then each path increases the flow at least by one unit, and augmentation still leaves behind integer capacities. By induction, after k iterations, the flow is greater than k. The algorithm terminates after $k = f$, and so the runtime is $O(mf)$, where f is the max flow. We can even trivially bound the maximum flow to be $O(mU)$, where U is the max capacity. Thus, the runtime is bounded by $O(m^2U)$.

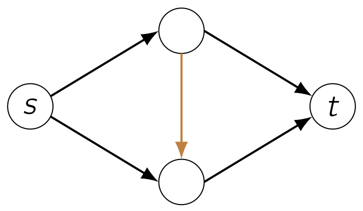
If G is a simple graph, meaning that we have no parallel edges, then the maximum flow is bounded by $O(nU)$ instead of $O(mU)$, reducing the runtime to $O(mnU)$.

With this algorithm we've actually proven the following:

Corollary 8.8 (Max Flow Integrality)
If G has integer capacities, then there exists an integral max flow.

Proof. Apply the above algorithm. At each step, the flow added to each edge is integral, and the algorithm eventually terminates. Thus, the resulting maximum flow is also integral. □

Though this runtime seems fine at first glance, it can actually be really bad since U is unbounded. For example, considering our previous example graph, let the brown edge have capacity 1, and the black edges have capacity 10^9.



It is not too hard to see that each augmenting path will only increase flow by 1, meaning that all 10^9 theoretical iterations will actually happen. Thus, even if capacities are integral, we can get theoretically infinite runtime!

This is actually even worse with rational values, because then our flow's guaranteed increase per iteration is only the product of the denominators of weights now.

This algorithm is worst with real numbers. It may take infinite time to converge to a solution, and may even converge to a non-maximum flow.

So, *we have an algorithm* (the **Ford-Fulkerson Algorithm**), but it sucks. We'll improve on it in the next lecture.

9 Lecture 9: Max Flows II

9.1 Improving on the Augmenting Paths Algorithm

Our previous algorithm had a runtime of $O(mnU)$. Is this runtime polynomial? Well, we usually define polynomial runtime to be polynomial in the size of the input itself. Since the capacity is exponential in the size of the input (since we only need $\log U$ bits to store it in the input), then our algorithm is not necessarily polynomial. Such situations, in which we have runtime polynomial in n, m but not in edge weight U, make up what we call **psuedopolynomial algorithms**.

If we run our algorithm for rational numbers, when we convert everything to integers by multiplying by the common denominator, we can create exponentially large runtimes. This shows that psuedopolynomial algorithms are not good at all!

Remark 9.1. *If we try to run the algorithm without multiplying out all the denominators, we still get the same runtime. The augmenting flows get smaller, but it still takes the same time to reach the maximum flow.*

Let's try to find a better algorithm for our problem. One thing to improve on in our previous algorithm is that we just chose any $s - t$ path we wanted. If we instead choose the path with **maximum bottleneck capacity** to augment our flow with, then our runtime should improve.

Can we find this path? Well, if we just examine all possible paths, then we have exponential runtime. A way to get around this is to just simply start removing edges (in the residual graph) in sorted order, and then check for connectivity. If s and t are still connected after removing each edge, then we can remove more; if they aren't connected after removing some edge, then we know that that edge was part of the bottleneck path. This is quadratic time, since we need $O(m)$ time to verify connectivity after each edge and $O(m)$ to iterate through all the edges. We can improve this runtime to $O(m \log m)$ if we remove half the edges at a time, and then binary search for the critical edge(s) instead. After we've found the critical edge(s), then we can simply use any standard search algorithm to find the $s - t$ path that goes through an edge of that capacity, which takes $O(m)$ time, so each iteration takes a total of $O(m \log m)$ time.

Remark 9.2. *We can also treat this problem as a directed minimum spanning tree problem, which means that we can apply our previous algorithms to improve this runtime to just $O(m + n \log n)$.*

Let's see if this new algorithm significantly helps our runtime. How many augmentations will we need in total? We have the path decomposition lemma again, which tells us that any flow can use at most m paths. Since we choose the maximum bottleneck path at every iteration, we know that our paths will have at least $\frac{f}{m}$ flow, where f is the residual flow. Thus, k iterations of max augmenting path tells us that we have $f(1-\frac{1}{m})^k \leq fe^{-k/m}$ remaining flow. Thus, if $k \geq m \log f$, then the remaining flow is less than 1, and by the flow integrality this means that we found the maximum flow. Thus, we will use $O(m \log m \cdot m \log f) = O(m^2 \log m \log mU) = \tilde{O}(m^2 \log U)$ time. This algorithm is just polynomial now, rather than psuedopolynomial, for both integers and rationals.

Note 9.3
We use the $\tilde{O}(f(x))$ notation to ignore logarithmic factors. Log factors don't matter since we aren't dealing with data structures anymore, except to theoreticians, which is why we use the $\tilde{O}$ notation.

9.2 Scaling Algorithms

We'll now go over another algorithm that has almost the same runtime, but includes a technique that doesn't rely on flow decomposition. This was developed by Gabow in 1985 and then Dinitz 1973, with Dinitz 'after' since his communication was not seen until after the Iron Curtain fell. The general idea is to apply a 'unit case' to a bigger problem.

Recall that our original algorithm took $O(m^2)$ time on unit capacity graphs. Thus, we can try to leverage this runtime, and we do so through **backwards rounding**. Essentially, we find the optimal solution to the rounded problem, unround, then fix up our solution.

For our problem, we'll first start by trivially 'rounding down' all our capacities c to $\left\lfloor \frac{c}{U+1} \right\rfloor$, where U is the largest capacity. To start off, all our capacities are rounded to 0, and the maximum flow is trivially 0. Our next step is to unround our capacities to either $0, 1$, and we do so with a bitshift - by repeatedly shifting in the most significant bit of our capacities. Every time we iterate, mathematically, we double all the capacities, and add 1 to some of these capacities. We can now find our new max flow by doubling the previous max flow, and then by using augmenting paths to find a new max flow.

How many augmenting paths will we need? Let's look at the residual graph, after doubling all the capacities. We know that for a $s-t$ cut, since we have a max flow already, that the edges crossing the cut will be saturated. When we add 1 to some edges, the number of edges crossing the cut with additional capacity 1 is at most m, and so the maximum flow increases by at most m after doubling. We can find one of these $s-t$ paths in $O(m)$. Thus, it takes us $O(m^2)$ time to do an update. This is a much simpler algorithm, and results in even a better runtime of $O(m^2 \log U)$!

Remark 9.4. *This type of bit-shifting actually sees quite widespread use, in computer architecture as well. In fact, one of Prof. Karger's interviews at Microsoft a long time ago relied on this.*

9.3 Strongly Polynomial Algorithms

Once again, theoreticians are never satisfied. The above algorithm may still fail to terminate for real numbers. Can we have a runtime independent of number sizes?

For our analyses, we will assume that we can now add/subtract numbers in $O(1)$ time. We now will look for a **strongly polynomial** algorithms, which take only polynomially many arithmetic operations in m, n. (Our previous algorithms were **weakly polynomial**).

Our previous algorithms depended on greedy algorithms, and so we would always have that logarithmic convergence to the max flow. The previous algorithms don't really give us ways to get rid of the logarithmic factor, and so we need a different measure of progress. One measure of being done is if s, t are disconnected in the residual graph. We can use this to come up with another heuristic - by looking at the number of edges in the shortest path connecting $s - t$. If this distance is infinite, then we have a max flow. If this distance is large, then intuitively means that we nearly have a max flow.

Thus, one thing we can try to do is to choose the shortest $s - t$ path at every iteration. We augment our flow with short paths, resulting in only long paths to remain, and so we may rapidly close in on a max flow. This leads to the **Shortest Augmenting Path Algorithm**, or **Edmonds-Karp Algorithm**, which works by simply finding the shortest augmenting path, and saturate it. We can use a breadth-first search to find a shortest augmenting path in time $O(m)$, and it takes $O(n)$ time to update the edges after we find this path.

We now claim that $O(mn)$ shortest augmenting paths will find the max flow, and so this algorithm is strongly polynomial takes $O(m^2n)$ time to find the maximum flow.

The reason that we don't only need $O(n)$ shortest augmenting paths is because some shortest augmenting paths may not change or even cause a decrease in the distance from s to t. Some experimentation shows that the latter doesn't happen, but we need to prove this, which we will do so in the next lecture.

10 Lecture 10: Max Flows III

10.1 Edmonds-Karp Algorithm

Let's now prove our claim from before, which will give us a strongly polynomial algorithm for max flows.

Theorem 10.1

After processing $O(mn)$ shortest augmenting paths we obtain the max flow.

Proof. To start, we'll introduce some notation: let $d(v)$ denote the distance from s to v in the residual graph. Now, we show a key lemma:

Lemma 10.2

If an shortest-path augmentation is performed, then $d(v)$ does not decrease.

Proof. Our proof will be by contradiction. We'll define $d'(v)$ to be the distance after augmentation, and so our condition for contradiction will be that there is some vertex such that $d'(v) < d(v)$. Note that the distance from $v = s$ to the source is always 0 and hence the lemma is true for the source.

Now, suppose that $v \neq s$ is the closest vertex to the source, and let P' be the shortest path to v, after augmentation. The path clearly exists since $v \neq s$. Since the path itself is nonempty, this means that there is a vertex w along this path immediately before v, which may possibly be s itself.

Then, $d'(v) = d'(w) + 1$. Since v was chosen to be the closest decreasing distance to the source after the augmentation, and $d'(w)$ is smaller, this means that the distance of w didn't decrease, or in other words, $d'(w) \geq d(w)$.

Now, this means that the edge (v, w) was created after the augmentation, because otherwise the distance to w would also have decreased. The only way we can create this edge is if our shortest augmenting path that pushed flow backwards, and so (v, w) was on the shortest augmenting path. This means that $d(v) = d(w) - 1 \leq d'(w) - 1 = d'(v) - 2$, a contradiction, and hence $d(v)$ never decreases. □

So now, we've shown that the distance of any node to the source never decreases in the residual graph with an augmentation. We now need to show that these distances will eventually increase.

Consider some shortest augmenting path that saturates the edge (u, v). If we augment with this path, then the edge (u, v) becomes removed from the residual graph, but another shortest augmenting path may bring it back, if the edge (v, u) is used.

When we use (u, v), we have that $d(v) = d(u) + 1$. When we bring back the edge by going through (v, u), we have that $d'(u) = d'(v) + 1$. By our lemma earlier, $d'(v) \geq d(v)$, and hence $d'(u) \geq d(u) + 2$. Or, in other words, every time we go through an edge and then bring it back, the distance of the shortest augmenting path increases by at least 2. Thus, after $O(n)$ saturations of (u, v), then $d(u) \geq n$, and no more shortest augmenting path will use (u, v). This means that after $O(mn)$ shortest augmenting paths, then there are no more paths to be added, and hence we've found a max flow. $\square$

Since we've proven the central claim, we now have shown that we can find max flows in $O(m^2 n)$ time.

Let's summarize our results on max flows so far:

- We have runtime $O(mn)$ on simple unit capacity graphs.
- We have runtime $O(m^2 \log U)$ on graphs with integer capacities.
- We have runtime $O(m^2 n)$ on general graphs.

It seems that as the graph gets more complex, the runtimes also become longer. But can we go faster?

10.2 Blocking Flows

The answer is yes, with a technique known as **Blocking Flows** to go faster. As algorithmists, we want to look at places where we are doing a lot of work, and see if we can increase our gain from the work. One place that we use a lot of work is when we are finding augmenting paths - after we add one augmenting path, we need to do our work all over again to find another augmenting path.

One way we can implement this in our previous shortest augmenting paths is to save the results of our breadth-first search. When we run the BFS, we can find many shortest paths, and we don't need to refind this after every augmentation. We don't need to rerun our BFS every time - we can simply use one, and then process all the edges that we have, since the distance cannot decrease!

Let's formalize this. We will first introduce some definitions, as follows:

Definition 10.3

We define an **admissible edge** to be one that goes forward one layer in a BFS. We define an **admissible path** to be one made of entirely admissible edges. We define the admissible graph to be one that includes all the admissible edges.

Now, we will formalize and say the following:

Lemma 10.4

Any admissible path is a shortest path.

Proof. Nonadmissible edges are either ones that go backwards or go through the same layer. They cannot go forward more than 1 layer, by the definition of BFS. If we have a shortest path that is nonadmissible, then it must have at least one nonadmissible edge. But this is a contradiction, since it will now take more steps to traverse to the sink t than an admissible path. □

Our idea is now to build an admissible graph, destroy all shortest paths, and then recompute until we find all augmenting paths. We'll define a **blocking flow** to be one using only admissible edges that saturates an edge on every admissible path. After we use a blocking flow, we don't make any new admissible edges (the only ones created are backwards edges), and there are no more admissible paths left in the residual graph. Hence, once we find the blocking flow, the distance from s to t has increased and we have to do a BFS again. The maximum distance is n, and hence we need at most $O(n)$ blocking flows to find a max flow.

Remark 10.5. *This also gives us a greedy approach to max flow. Each blocking flow doesn't make any new paths, and so we can measure progress towards actually finding max flows.*

Let's now try to find a good way to find blocking flows. We'll first start by analyzing the unit capacity case. Given an admissible graph, we start at s, continue along admissible edges until we get to some vertex v, and keep advancing forwards until we either reach t or cannot advance from v. If we reach t, this means that we found an augmenting path, and so we saturate it and remove all edges, and restart the algorithm to continue finding more augmenting paths in the admissible graph. Otherwise, if we have nowhere to go from v, this means that going to v is useless and we can delete the edge (u, v), and continue the search from v. We continue until we are blocked at s, which means that there are no more paths from s to t.

Let's now look at runtime. The number of retreats is $O(m)$, since each edge can be deleted at most once. Each edge is also saturated at most once (using the unit capacity assumption here), for a total time of $O(m)$. Finally, since the number of advances is equal to the number of saturations plus the number of retreats, this is also $O(m)$. Thus, the total runtime to find a blocking flow is $O(m)$, and so the total runtime for finding a max flow is $O(mn)$. This is a slight improvement over our original algorithm for unit capacity graphs, since this analysis now applies to non-simple graphs.

This is not very exciting in itself, but it can be used to actually get a better bound for unit capacity graphs and also generalize to non-unit capacity graphs. For the unit capacity case, let's suppose that we find k blocking flows, and so the distance from s to t has distance at least k. By flow decomposition, each remaining (disjoint) $s-t$ path uses up at least k edges, and sends at least k flow. Thus, the number of remaining paths that we have is $\leq \frac{m}{k}$. Since each blocking flow we do finds at least one path, this means that $\frac{m}{k}$ additional blocking flows finish the algorithm. The total runtime is then $O(km + \frac{m}{k}m)$, which when balanced, shows that the runtime is actually $O(m^{3/2})$. This is the same runtime as previously in dense graphs but much better in sparse graphs.

Remark 10.6. *Using this technique, we can also show with a more sophisticated analysis that we get time $O(mn^{2/3})$ in simple unit capacity graphs and $O(mn^{1/2})$ in bipartite unit capacity graphs.*

Let's now generalize to graphs with capacity. The number of advances still is bounded by the sum of the other terms, not affecting our runtime. We still have $O(m)$ retreats. However, for augmentations, we may only at most destroy one edge. We still need to perform $O(m)$ augmentations, but now we have $O(n)$ work per edge, for a total of $O(mn)$ time. Thus, the time it takes for a blocking flow is $O(mn^2)$, which is still better than the shortest augmenting paths algorithm. The intuition for the improvement is that an augmentation only costs $O(n)$ rather than $O(m)$.

We'll now refine the analysis here. If the value of the max-flow is f, then the augments at most cost nf. Our other work needed is $O(m)$ per blocking flow, and so our runtime is $O(mn + nf)$. If we can ensure a small flow, then our runtime will be fast. We can do this by the same scaling algorithm as before. Each scaling step added at most m residual flow, and thus the runtime becomes $O(mn \log mU)$, which is better than before.

We now return to the central question - *can we do better*? Yes! Over the period from 1985 to about 2010, we got a strongly polynomial runtime of $\tilde{O}(mn)$ with the push-relabel algorithm, then $O(m^{3/2} \log U)$ by Goldberg and Rao with more sophisticated scaling techniques. After 2010, algebra people (including fellow MIT professors Kelner and Madry) came and used matrix algebra to reduce the running time to $O(m^{10/7} \log U)$ and now are pretty close to $O(m^{1+\epsilon} \log U)$.

Remark 10.7. *Push-relabel commonly achieves near-linear time, and so is a recommended algorithm whenever a max flow comes up.*

Joke 10.8. *These algebraic techniques 'destroy all the combinatorical structure' of the problem, and so anyone who matches these bounds with combinatorical techniques gets an A.*

11 Lecture 11: Max Flows IV

11.1 Improving Blocking Flows

Let's go back to our blocking flow algorithm, and see if we can improve it. Our slowest step with the blocking flows algorithm was the augmentation step, and we can try to use a clever data structure to speed it up. Tarjan did exactly this and introduced the **link-cut tree** data structure to do so. Essentially, we have a directed forest of positive capacity edges, and we want to implement the augment/retreat/advance operations quickly. One of the operations that we want to introduce is the *advance-to-root* operation, which simply jumps from one vertex to the root of that tree. Then, we check if the root has any edges of positive capacity to another tree. If it does, then we've just discovered a longer path, and so we **link** the root of the tree to another. We continue this process until we reach the vertex t, at which point we've found a path of nonzero capacity which we can use to augment our flow with. Finally, since at least one edge is going to be saturated after augmentation, then we can **cut** the tree there and repeat.

Tarjan's implementation uses Splay Trees and can perform the link, advance, cut, and augment operations in amortized $\log n$ time, and so our max flow algorithm now will have runtime $O(nm \log n)$, a significant improvement. $O(mn \log_{m/n} n)$ is also possible with some tweaks.

Remark 11.1. *Though the link-cut tree seems like a collection of random operations made specifically for max flow, it actually has decent applicability outside of this problem as well.*

11.2 Min-Cost Max Flows

We now know that we can relatively efficiently find a max flow, but we still have the question of choosing a 'best' max flow. To answer this, we'll add a cost $c(e)$ on every edge e, and say that the best max flow is one that minimizes $\sum c(e)f(e)$. In this problem, the **min-cost max flow** problem, the edge weights can also be negative. When we construct the residual graph with an edge (u, v) having cost $c(e)$, then the cost of (v, u) should logically be $-c(e)$.

Let's first consider different variants of this problem and see that reductions are possible. One variant of this problem asks for the min-cost to send a flow v through the graph. We can reduce this problem to the general min-cost max flow problem by simply introducing a bottleneck edge of capacity v at the source, ensuring that the max-flow we find is v. In addition, the shortest path and max flow problems can easily be reduced to this problem.

Another variant is the **minimum-cost circulation** problem, which tries to find a flow with everything balanced, and the condition that we can have negative weight edges make this problem interesting.

We can reduce to the min-cost max flow problem by simply not connecting s or t to any nodes, since the max flow is trivially 0 and a min-cost circulation will be induced in the graph. We can reduce the other way by adding an ∞-capacity edge from t to s with $-\infty$ cost, since we will send as much flow as possible from t to s to minimize costs, and so the max flow goes to t. These infinities can simply be represented by mU and $-nC$, where respectively U is the max capacity edge and C is the max cost edge. The reason why $-nC$ is sufficient is since a circulation can go through n edges, and so the total forward cost is at most nC, making it profitable to send flow.

Another reduction method to min-cost circulations starts instead by finding any max flow f (with previous algorithms) in our graph. We want to find the min-cost flow f^*. Now, $f^* - f$ is a circulation, since there are no more $s - t$ paths and the net flow is zero. Though this new flow may not be feasible in the original graph, this flow is feasible in the residual graph G_f. If $f_e^* - f_e \geq 0$ then we have a positive flow, and the amount of the flow is $\leq u_e - f_e$, which is the capacity of edge e in G_f. If $f_e^* - f_e < 0$ then we have a reverse direction flow of value $f_e - f_e^*$, which is less than the reverse edge capacity of f_e. Thus, this is a valid flow in the residual graph. We can then find a min-cost flow f^* by first finding the min-cost circulation q in the residual graph, and then adding it to f.

Remark 11.2. *While we previously looked at net-flow, the flow going from one vertex to another, we're now instead going to focus on the flow per edge. This has the benefit of allowing multiple edges with different capacities and costs going between the same vertices vertices.*

Let's now look at deciding optimality. How can we verify if f is either a min-cost flow or not? We can verify if f is a max flow easily by a min-cut, as seen previously. We can verify f being a min-cost flow if and only if the min-cost circulation in G_f has cost $= 0$. If a negative circulation exists, then it can be added to our flow to reduce the overall cost, but this is kind of cheating since we can't find a min-cost circulation easily yet.

A way around this is to instead consider the cycle decomposition of the min-cost circulation. When it is negative, at least one of the cycles made by the min-cost flow has to have negative cost. We can then add this cycle to f for a smaller cost flow, and so a negative circulation implies a nonoptimal flow. The converse is also seen to be true, since a nonoptimal flow means that there must be at least one negative cost cycle. Thus, checking for negative cost cycles gives us a way to verify that we have a min-cost flow.

We can find negative cost cycles with the **Bellman-Ford** or the **Floyd-Warshall** shortest path algorithms (running in time $O(mn)$). Shortest paths are not well defined when we have negative cycles, but these algorithms can detect the negative edge cycles. We can use this to find a min-cost-circulation by running on G_f, then saturating the cycle, then repeating. This algorithm, known as **Klein's algorithm**, is similar to Ford-Fulkerson and takes $O(mCU)$ iterations to finish. It sucks.

12 Lecture 12: Max Flows V

We got *an algorithm* for the min-cost flow problem, based on cycle cancelling. While our previous implementation sucked, there are actually ways to make cycle cancelling polynomial time (similarly to how the bottleneck paths algorithm changed Ford-Fulkerson). An improvement on cycle cancelling will be explored on the homework, but let's explore a different idea now.

12.1 Price Functions

Suppose that we have some cities, for which there is infinite supply of tequila at the source s, and infinite demand for it at the sink t. Each road connecting cities will have some cost of transportation, and we want to find the max flow at min cost. If we were to just mandate how much flow goes where, then this is a command economy, not acceptable in America.

So, let's instead think about a capitalist approach. We won't demand global optimization, but rather see what free merchants will do. Each merchant will pick up the tequila at s and deliver it to other neighbors of s. Though there is no demand for tequila at the neighbors, they can still sell the tequila to other merchants along the route. This creates a free market, from which each vertex will now have a price $p(v)$ at which tequila is bought and sold.

Now, suppose that a merchant is going from v to w. The net profit that they gain per unit transported is $p(w) - p(v) - c_{vw} \geq 0$ which is nonnegative in order for the prices to be feasible. Thus, this gives us a motivation to define a reduced edge weight as $c'_{vw} = c_{vw} + p(v) - p(w)$. Then, our condition for making a profit is $c'_{vw} \leq 0$.

If there is a profitable nonzero cost then, the demand at v will increase and the supply at w will increase, increasing the price at v and decreasing the price at w, by basic principles of economics. This represents some instability in the market, and is self-reinforcing, and can only stop either when the edge itself is saturated or when the reduced cost becomes positive.

This observation allows us to define feasibility as follows:

Definition 12.1

A price function is **feasible** for the residual graph if and only if no residual edge has negative reduced cost.

We also have a very important lemma about price functions:

Lemma 12.2

Using a price function doesn't change any costs of cycles.

Proof. $\sum c'_{vw} = \sum (c_{vw} + p(v) - p(w)) = \sum c_{vw}$ which telescopes since the start and end vertex is the same. Thus, no matter what price function we use, the cycle costs remain the same. $\square$

Since the costs on a cycle don't change, then as we saw before, a negative-cost cycle means that the flow is not yet optimal. This doesn't change no matter what price function we have.

Now, we have a central claim that shows the utility of price functions:

Theorem 12.3
A circulation/flow is optimal if and only if there is a feasible price function on its residual graph.

Remark 12.4. *This also allows for a feasible price function to serve as a certificate.*

Proof. To show that feasible prices lead to optimality, we simply need to show that feasible pricing leads to no negative cycles, as seen earlier. When we have a feasible price function, we have no negative reduced cost residual edges, and hence there are no negative reduced cost cycles, which means there are no negative cost cycles (by the previous lemma), and hence the flow is optimal.

For the other direction, we need to show that an optimal min flows means that we can construct a feasible price function p on the residual graph. In other words, there is no way for merchants to make money. Now, we claim that $p(v)$ being equal to the shipping cost to v from s is feasible. This doesn't cover all vertices, since some may be isolated in the residual graph, but we can elegantly get around this by changing our source to s', which will have 0 edges to every vertex. Then, we just let $p(v)$ be equal to the shortest path cost from s' to v, which we can compute since there are no negative cycles, as our flow is optimal.

The only thing left is to check feasibility. We need to verify $c'_{vw} \geq 0 \iff p(v) + c_{vw} \geq p(w)$, and this holds by the triangle inequality on shortest paths. $\square$

If we find a feasible price function, then we can find a min-cost flow. Let's consider the effect of increasing the price at one vertex. This decreases the incoming edge costs and increases the outgoing cost by the same amount. This doesn't change any of the costs of paths through the vertex, but can eliminate the negative edge. Thus, if we tweak the prices at each vertex, we should be able to find a feasible price function now.

12.2 Min-Cost Flow Algorithms

Our previous algorithm for min-cost flows was to start with a max-flow, and then make it optimal by cycle-canceling. Here, instead, we'll start with an empty flow, and then use augmenting paths to increase the flow while at the same time keeping the min cost. Every augmenting path that we choose should be greedy, in order to minimize cost, and should be the shortest augmenting path (with respect to costs, rather than distance).

Now let's show that this is a good, working algorithm. We just want to show that the cost is minimized at all time, which means there are no negative cycles in the residual graph.

Theorem 12.5
Adding a shortest augmenting path to a graph with no negative cost cycles in the residual graph leads to a residual graph with no negative cycles.

Proof. Let's proceed by contradiction. The only way for a negative cost cycle to be created in an augmentation is if that augmentation reversed an edge along the path. But we can simply cancel out the forwards edge with the backwards edge, removing the cycle and creating a new path. This is a new path with lower sum of costs, and hence when a negative cost cycle is created, this means that a shortest augmenting path was not used. □

Let's also see an alternate proof of this theorem. The key idea behind this proof is that the price function shifts all $s - t$ paths by the same amount, which is $p(t) - p(s)$, by telescoping.

Proof. If we have no negative cost cycles, then we can compute shortest paths from s to define new prices. Then, the edges on all shortest $s - t$ paths have reduced cost 0. The reduced cost of the reverse edge is then likewise 0 on this graph. If we augment along such a shortest path, then they will have 0 cost and create 0 cost reverse edges. Thus, no negative cost cycles can be created, and we still have a feasible price function. □

Since we've proven correctness, let's now look at runtime. Each augmentation finds at least one unit of flow, and then we have a min-cost max flow after f iterations. Each shortest-path finding is $\tilde{O}(m)$ by Dijkstra's, and hence the total runtime is $\tilde{O}(mf)$, just like the original Ford-Fulkerson algorithm.

Dijkstra's algorithm here requires non-negative edges. If we start with a graph of non-negative edges, run Dijkstra's and then augment, using the reduced edge lengths afterwards. Then, by feasible pricing, these reduced edge lengths are always positive, and so we don't run into problems.

This algorithm has two important limitations, which are that we can't deal with negative costs just yet, and that the algorithm is only psuedopolynomial, but both can be resolved relatively easily, as we will see next time.

13 Lecture 13: Max Flows VI, Linear Programming I

13.1 Polynomial Min-cost Flow

We now have a psuedopolynomial min-cost flow algorithm, and now let's try to make this polynomial. We can do this, as we did for the general max flow problem, with scaling techniques on the capacities themselves. We shift in one bit at a time, as before, creating some residual flow, which we can eliminate with more augmentations, and this runtime is the same as before, $O(m^2 \log U)$.

However, we have a problem with the shifting method. A 0-capacity edge can potentially become a 1-capacity edge, which can result in an edge with negative cost being made, since our previous algorithm only guaranteed non-negativity on edges with nonzero capacity.

We can resolve this by first sending flow on all negative arcs. This makes all costs greater than zero, but violates flow conservation, making some nodes have deficits and others have excesses. Next, we send excesses to the deficits themselves. This is another min-cost flow problem, but now, we don't have any negative arcs! So, we can compute a fast min cost flow (actually a circulation) and a price function with Dijkstra's, resulting in a residual graph with no negative arcs. This is an algorithm that actually computes min-cost circulations, for any graphs. This gets rid of our second issue with max flows as well.

So, if we use the above algorithm in our scaling algorithm, we can use a min-cost circulation to balance the excesses and deficits. This takes runtime $\tilde{O}(mf) = \tilde{O}(m^2)$. When we take care of this issue in scaling, our final runtime is $\tilde{O}(m^2 \log U)$.

Can we do better? The previous algorithm we described is known as "capacity scaling" for obvious reasons, but there are also "cost scaling" algorithms which have a $\log C$ term rather than a $\log U$ term. In fact, strongly polynomial algorithms do exist, but are too much work to describe here. Tardos '85 proposed one based on scaling, but said that the small bits are negligible, and thus the scaling techniques don't need to contain a $\log U$ term. Currently, our best runtimes are $\tilde{O}(m^2)$. The best scaling algorithms result in $\tilde{O}(mn \log \log U \log C)$ runtime.

13.2 Complementary Slackness

Let's go back to our merchant analogy, when we previously considered reduced costs, and that everything was in equilibrium. If a reduced cost is positive, then there will be no flow, since otherwise we would ship at a loss. If a reduced cost is negative, then that edge will be saturated. Thus, all the reduced costs must be equal to zero in equilibrium.

This property is known as **Complementary Slackness** and in fact is a criterion for optimality. This also suggests another algorithm for finding a min-cost flow - saturate all the negative cost arcs, send the flow back using only zero arcs (since positive arcs cannot be used at all). The cost of any path is zero, which just turns the min-cost flow problem into a standard max flow problem! So, in a general sense, a min-cost flow problem is just a shortest paths problem combined with a max flow problem.

Remark 13.1. *With complementary slackness, we see that any feasible price function is going to make a set zero-cost edges, which are the important edges to optimize. Network simplex algorithms work explicitly with these zero-cost and are another way to solve min-cost flow.*

13.3 Linear Programming

Consider the min-cost flow problem, in a purely algebraic presentation: to minimize the quantity $\sum c(e)f(e)$ in where $0 \leq f(e) \leq u(e)$ and $\sum_{e\,in\,v} f(e) = \sum_{e\,out\,v} f(e)$. Note that all the equations have a linear objective function and the constraints are all linear inequalities. Linear Programming is the study of such problems. Dantzig, in the 1940s, designed the Simplex Algorithm for solving such problems, which is still the preferred algorithm for solving linear programming problems. The problem also established the field of combinatorial optimization, in that other combinatorial problems such as shortest paths and max flow can be formulated as linear programs.

While Dantzig had an algorithm, the theory itself was lacking behind. We couldn't answer the questions of whether solutions exist, whether the solution could be written (rational solutions), or whether the solution's optimality could be checked. Dantzig showed that linear programming is in NP, but didn't make much progress on finding a polynomial algorithm for solving it. It wasn't until the 1970s that the **ellipsoid method** was developed and proven to be polynomial by Khachian (though this was still not practical). In 1983, Karmarker introduced the interior point method, which is polynomial and practical.

The general form of a **linear program** involves variables and constraints (linear equalities or $\leq$ and $\geq$ inequalities). We call a vector $\vec{x}$ **feasible** if it satisfies all constraints, and we call the linear program **feasible** is there is a feasible $\vec{x}$. We call an $\vec{x}$ **optimal** if it leads to the best objective function value over all feasible points. Finally, we call a linear program **unbounded** if there are feasible $\vec{x}$ of arbitrarily good value.

Just from the definitions introduced, we can already show an important lemma:

Lemma 13.2
Every linear program is unbounded, infeasible, or has an optimum.

Proof. This follows from the compactness of $\mathbb{R}^n$, where we try to keep finding series of optima. Either we find no limit, in which case the linear program is unbounded, or we find a limit, which we converge to. $\square$

We can express a linear program in **canonical form** as follows:

Definition 13.3 (Canonical Form)
The goal will be to maximize the quantity $c^\intercal \cdot \vec{x}$, subject to the constraint that $A \cdot \vec{x} \leq \vec{b}$, where A is our matrix and the inequality sign is taken coordinate-wise. This matrix multiplication is a shorthand way of saying that if we have rows a_i, then $a_i \cdot x \leq b_i$.

Lemma 13.4
Every linear program can be transformed into one in canonical form.

Proof. We will show that any deviation from canonicality can be fixed, as follows:

- If our linear program asks us to minimize some quantity, we can simply maximize the negative of that quantity.
- If our linear program has inequalities not of the form $\vec{a_i} \cdot \vec{x} \leq b_i$, then we can simply rearrange our equation to do so.
- If our linear program has $\geq$ inequalities, then we can simply multiply the inequality by -1.
- If we have an equality, then we just introduce two inequalities, of the form $\geq b$ and $\leq b$. $\square$

We also have a related form, called **standard form**, defined as follows:

Definition 13.5 (Standard Form)
The goal will be to minimize the quantity $c^\intercal \cdot \vec{x}$, subject to the constraint that $A \cdot \vec{x} = \vec{b}$, and $\vec{x} \geq 0$, where as before where A is our matrix and the inequality sign is taken coordinate-wise.

Lemma 13.6
Every linear program can be transformed into one in standard form as well.

Proof. We'll show that every canonical form linear program can be transformed into standard form. Transforming the max to a min is easy. To change an inequality to an equality, we use a trick known as **slack variables**, where we add a variable s_i to change the inequality $\vec{a_i} \cdot \vec{x} = b_i$ to an equality $\vec{a_i} \cdot \vec{x} + s_i = b_i$. Finally, we will introduce positivity by introducing two variables $x^+ \geq 0$ and $x^- \geq 0$. Then, we just define $x = x^+ - x^-$ which changes the program into a standard form one. $\square$

14 Lecture 14: Linear Programming II

Now that we have defined a linear program and seen some reductions to the standard and canonical forms, let's investigate the structure of the solutions. Specifically, we will check if there exists a solution, and how we can verify the optimality of it.

14.1 Verification and Size of Solutions

To answer these questions, let's first consider the case of linear equalities, of the form $A\vec{x} = \vec{b}$, in the case of a square matrix A. A solution $\vec{x}$ can easily be verified here simply by computing $A\vec{x} = \vec{b}$. There are many criteria regarding the nature of solutions, which is summarized below:

Claim 14.1

The following are equivalent:

- A is invertible
- $A^\intercal$ is invertible
- $\det A \neq 0$
- A has linearly independent columns or rows
- $A\vec{x} = \vec{b}$ has a unique solution for all $\vec{b}$
- $A\vec{x} = \vec{b}$ has a unique solution for some $\vec{b}$

Proof. See any standard linear algebra text. □

Question 14.2. *Can we actually write $\vec{x}$ down? Specifically, we cannot do so if some components of $\vec{x}$ are irrational.*

Well, let's consider how much space we need to write down a number n. For an integer n, we use $\log n$ bits to write it down. If we have a rational number $\frac{p}{q}$, then the number of bits is $\log p + \log q$. When we multiply two numbers, then their size is equal to the sum of their individual sizes. An n-vector size equal to the sum of the sizes of the numbers in the vector, plus $\log n$ to indicate the size of the vector. A $m \times n$ matrix has size equal to the sum of the entries in the matrix, or mn times the size of the largest entry. The size of the matrix product is less than the sum of sizes of the multiplied matricies themselves.

Our goal is for finding a solution polynomial in the size of the input. Obviously, if we cannot write down $\vec{x}$ with the size of an input, then doing so is impossible!

Theorem 14.3

For a square matrix A, $\det A$ is polynomial in the size of A.

Proof. Let S_n denote the symmetric group, whose elements σ are the permutations on n elements. We define the sign of σ, as usual, to be equal to 1 if σ is in the alternating group A_n, and -1 otherwise. Use the formula for the determinant

$$\det A = \sum_{\sigma \in S_n} \operatorname{sgn}(\sigma) \prod_{i=1}^{n} A_{i,\sigma(i)}$$

The inner product itself is the product of n numbers, which has size at most S^n, where S is the value of the largest element. There are $n!$ permutations, and so the maximum value of the determinant is at most $n!S^n$. The size of the determinant is then $\log n!S^n = O(n \log n) + O(n \log S)$ which is polynomial in $\log S$ and n. □

Corollary 14.4

The inverse matrix can be computed with polynomial size, and hence a solution can be written down in polynomial size.

Proof. Recall from linear algebra that $A^{-1} = \frac{1}{\det A}$ times the cofactors of A. Each entry of the inverse matrix $A_{ij}^{-1} = \frac{\det A_{ij}}{\det A}$ is polynomial size, and hence the inverse has polynomial size. Once we have the inverse (either through direct computation or Gaussian elimination, which can be shown to use polynomially many operations), we can get our solution in polynomial size. □

Remark 14.5. *Even in a matrix whose entries are mostly zero, the size of the determinant and hence the inverse can be $O(n \log n)$ bits. That's a lot, but still polynomial in the input size.*

Now that we've shown that solutions to the square matrix case can be found and written down, let's look at non-square matricies. When we have the equation $A\vec{x} = \vec{b}$, this means that the columns of A span $\vec{b}$. If we can find a maximal linearly independent subset of these columns which span $\vec{b}$, we can augment this set to form a basis of vectors, from which we can construct a square matrix and hence check a solution in polynomial time as per above, since any solution has to zero out the added basis columns.

So, we have that the solution $\vec{x}$ can be written down for any matrix A in polynomial space and thus it can serve as a witness for solvability. Can we now check that we have no solution to a system in a easy-to-verify manner? We need to show that the columns of A don't span $\vec{b}$, or in other words, show that $\vec{b}$ is not in the span of the columns of A, the set $\{Ax | x \in \mathbb{R}^n\}$.

Let's first consider the 2D case. The span of A is either the entire plane, a line, or the zero vector. Only the line case is interesting. If b is not in the subspace defined by the line, then b can be decomposed into the sum of a vector parallel to the line and a vector perpendicular to it. We can use the perpendicular vector y as a witness then - simply showing that $y^\intercal A = \vec{0}$ but $y^\intercal \vec{b} \neq 0$ will suffice.

How do we actually find y satisfying these equations? We can either do fancy math with projections, or be smart. Notice that if $y^\intercal \vec{b} \neq 0$, then we can scale y appropriately such that $y^\intercal \vec{b} = 1$. Obviously, scaling y doesn't change the veracity of the first equation. Then, we just need to solve

$$y^\intercal \cdot \left(A \mid b\right) = \left(\vec{0} \mid 1\right)$$

which is just a system of linear equations that we know how to solve. This also tells us that the size of y is polynomial in A and b, which is great!

Now we generalize to the n-dimensional case.

Theorem 14.6
$A\vec{x} = \vec{b}$ has no solution if and only if there is some y such that $y^\intercal A = \vec{0}$ but $y^\intercal \vec{b} \neq 0$.

Proof. The proof from the 2-dimensional case for the forward direction still holds, in that if some y exists, then the equation has no solution, since y is outside the span of A. Now, for the opposite direction, if $A\vec{x} = \vec{b}$, then $y^\intercal A\vec{x} = y^\intercal b$. However, the left hand side is zero but the right hand side is nonzero, implying that no solution exists. $\square$

Remark 14.7. *These two linear systems provide a sort of duality, in that a solution not existing in the second means that a solution exists to the first, and a solution existing in the first means that there is no solution to the second.*

Joke 14.8. *This is the only lecture in the course that is math-heavy and devoid of algorithms.*

14.2 Geometry of Solutions

Let's now consider the geometry of the inequalities that we have. While we would love to visualize n-dimensional space, this is currently impossible.

For the 2D case, when we plot all the inequalities, we have many lines, such that any solution must either be above or below the lines as necessary to satisfy the linear program. The end space is a polygon within the enclosed space itself, such that a point in the polygon itself is a solution, while one outside of the polygon is not. This generalizes to n dimensions, where we make finitely many

halfspaces from our inequalities, ultimately creating a **convex polytope**. To show convexity, it is enough to note that two points on one side of a halfspace will have their connecting line also on the same side.

Definition 14.9

A **polytope** is the space corresponding to the union of finitely many halfspaces. A polytope is **convex** if for any two points on the polytope, the line connecting them is also entirely within the polytope itself.

Now let's consider where the optimum of our linear program is found.

Claim 14.10

The optimum can be found at some corner of the polytope.

Intuitively, moving in the same direction as the objective function will increase the value. We will eventually hit one of the halfplanes defining the polytope, after which we can keep following this border until we hit another halfplane at a corner that prevents us from further improving our solution.

To prove this formally, let's define some terms:

Definition 14.11

A **vertex** is a point which is not a convex combination of two other points.

A **extreme point** is a point that is a unique optimum solution for some objective c.

A constraint $ax \leq b, ax = b, ax \geq b$ is **tight** when $ax = b$.

A point is **basic** if all the equality constraints are tight and if n linearly independent constraints (including the equalities) are tight.

A point is a **basic feasible solution** if a point is basic and is feasible in the linear program.

Now, we claim that the definitions of vertex, extreme point, and basic feasible solution are all equivalent.

First, we show some lemmas:

Lemma 14.12

In any standard form feasible linear program, the optimum appears at a basic feasible solution.

Proof. Suppose that we have an optimum that is not a basic feasible solution. This implies that there are less than n tight constraints, which means that there is a degree of freedom to move $\vec{x}$

around while keeping the tight constraints. More formally, we have a non-zero-dimensional subspace such that $\vec{x}$ satisfies all the constraints.

If we move $\vec{x}$ around, we either can improve our solution, worsen our optimum, or leave it unchanged. The first possibility is impossible by our assumption, and likewise the second is impossible since we improve our solution by moving in the opposite direction, which is impossible. Thus, the only possibility is that the entire subspace is optimal.

For a general polytope, the entire subspace being optimal doesn't mean that a basic feasible solution exists there. However, imposing standard form does.

Currently, our subspace contains a line through the current optimum $\vec{x}$. We can write this line $\vec{x}+\epsilon\vec{d}$ for some direction $\vec{d}$, and we move along this line until some x hits zero. This makes a new tight constraint. Then, repeat this enough times until x becomes a basic feasible solution. □

The proof above leads to the direct corollary:

Corollary 14.13

If there is an $\vec{x}$ of a given value, then there is also a basic feasible solution no worse than $\vec{x}$.

Remark 14.14. *Practitioners like standard form, since it is useful, as shown above. Theoreticians like canonical form, since it is pretty.*

We will continue this proof and relate extreme points and vertices to basic feasible solutions in the next lecture.

15 Lecture 15: Linear Programming III

15.1 Equivalence of Definitions Continued

We continue our proof from last time.

Lemma 15.1

The definition of a basic feasible solution and a vertex are equivalent.

Proof. To show this, we first suppose the contrary, that a basic feasible solution is not a vertex. Then, the solution is a convex combination of two points in the polytope, which is a line in the polytope through the vertex, which are all feasible. We thus cannot have n tight linearly independent constraints at the point, which means that this is not a basic feasible solution, which is a contradiction, and thus a basic feasible solution is a vertex.

For the other direction, if we don't have a basic feasible solution, then we have less than n tight constraints. They then define a feasible subspace of positive dimension, which creates two points whose convex combination is the point, i.e the point is not a vertex. Thus, a vertex is a basic feasible solution. □

Now, let us show the equivalence of extreme points and basic feasible solutions.

Lemma 15.2

The definition of an extreme point and a basic feasible solution are equivalent.

Proof. As before, we will suppose the contrary. Assume that an extreme point is not a basic feasible solution. If it is not, then we have less than n tight constraints, which means we have a feasible line through the solution. The only possibility for an optimum here is to leave the objective function unchanged (as we saw before), which contradicts our definition of extreme point. This means that all extreme points are basic feasible solutions.

For the other direction, we need to show that a basic feasible solution is an extreme point. To do so, we assume standard form, but this easily generalizes. Let T be the set of tight $x_i \geq 0$ constraints. Define the objective to minimize $\sum_{x_i \in T} x_i$, which obviously has a minimum at 0. Now, consider any other feasible point. It's loose on some $i \in T$, which means that the objective value is not optimal. Thus, the basic feasible solution is the unique optimum, and hence is an extreme point. □

Theorem 15.3
The definitions of vertex, extreme point, and basic feasible solution are equivalent, at least for standard form LPs.

Proof. This follows immediately from lemmas 14.12, 15.1, and 15.2. □

Now that we have these proofs of optimality, we have a simple algorithm for solving linear programs: check all basic feasible solutions. There are $\binom{m}{n}$ finite basic feasible solutions, which is exponential in the size of our input, but finite. To actually try all of these, we can solve the linear equations with standard Gaussian elimination or inverse multiplication. From this, we can also conclude that any optima has polynomial size.

15.2 Complexity

Let's now look at the linear programming as a decision problem for now. We want to find the complexity of answering if the optimum of the linear program is greater than some constant k. Obviously, this is verifiable in polynomial time, and so linear programs are in the class NP. However, is there a way to show that the optimum is less than the constant k (which will show that LP is in co-NP)?

15.3 Duality

We can show this with the concept of **duality**, which is a central concept in combinatorial optimazation. It will give us a succint proof that we can't do better than some bound, given that the point is optimal. We previously saw special cases of duality before, in the max-flow min-cut theorem, the prices of min-cost flows, the potentials of shortest paths, and in Nash equilibria in 2 player games.

In a linear program in standard form, asking us to minimize $c^\intercal x$, satisfying $Ax = b$ and $x \geq 0$. Suppose v^* is the smallest value of the objective. How do we get a lower bound on v^*?

We can multiply the constraints by some constants, then add them together. More specifically, we multiply each equality $a_i x = b_i$ by some y_i, then add all the equations together. We get that the sum of all constraints is the equation $y^\intercal Ax = yb$. If we find some $\vec{y}$ such that $y^\intercal A = c^\intercal$, then $y^\intercal b = c^\intercal x$, for every feasible x, then all feasible points have the same objective value! Thus, the only linear programs for which such $\vec{y}$ exists are silly ones, and thus any nontrivial linear program does not admit a $\vec{y}$.

This is too restrictive. Let's instead try to relax our constraints and find a $\vec{y}$ such that $y^\intercal A \leq c^\intercal$, coordinatewise. This tells us that $y^\intercal b \leq c^\intercal x$, due to the nonnegativity constraint on x_i, showing that we have a lower bound on the value of the optimum of $y^\intercal b$. Thus, maximizing $y^\intercal b = b^\intercal y$ satisfying $y^\intercal A \leq c^\intercal \iff A^T y \leq c$ gives us a tight lower bound on the system! We call this new linear program the **dual program**, and call the original one a **primal program**.

In short, we have shown the following:

Theorem 15.4 (Weak Duality)
Let v^* be the optimum for a linear program in standard form, and let w^* be the optimum for the dual program. Then $v^* \geq w^*$.

As a sanity check, let's show that the dual of a dual LP is primal. A dual linear program is one where we need to maximize $y^\intercal b$ such that $y^\intercal A \leq c^\intercal$. Converting to standard form, we need to find $\min -b^\intercal y$ subject to $Ay + Is = c$. Letting $y = y^+ - y^-$, then our objective is to minimize $-b^\intercal y^+ + b^\intercal y^-$ such that $Ay^+ - Ay^- + Is = c$ where all the vectors $y^+, y^-, s \geq 0$. We've now converted to standard form; writing this out in matrix form, our goal is to minimize the quantity

$$-\begin{pmatrix} -b^\intercal & b^\intercal & 0 \end{pmatrix} \begin{pmatrix} y^+ \\ y^- \\ s \end{pmatrix}$$

subject to the constraint that

$$\begin{pmatrix} -A & A & I \end{pmatrix} \begin{pmatrix} y^+ \\ y^- \\ s \end{pmatrix} = \vec{c}.$$

If we take the dual of the (now-standard-form) dual, then our goal is to maximize $c^\intercal z$ where z satisfies

$$\begin{pmatrix} A & -A & I \end{pmatrix} z \leq \begin{pmatrix} -b^\intercal & b^\intercal & 0 \end{pmatrix}.$$

The first two constraints means that $Az = -b^\intercal$. The third implies that z itself is componentwise negative, and hence $-z$ has all positive entries. Letting $x = -z$, then our goal will be to minimize $c^\intercal x$ where $x \geq 0$, satisfying $Ax = b$, which is a LP of primal form!

Remark 15.5. *The dual of a standard form linear program is a canonical form linear program. So, both forms are necessary in the description of linear programs.*

Theorem 15.6 (Duality Feasibility Relations)
Sometimes linear programs are unfeasible or unbounded. We actually only have a few cases relating the primal and the dual:

- Both programs are bounded and feasible.
- One is infeasible, and the other is unbounded.
- Both are infeasible.

Proof. There are three possible cases for each program (feasible bounded, feasible unbounded, infeasible), but we can show that out of the $3^2 = 9$ possible cases, only the four listed above are actually possible.

Suppose we have a primal program P and a dual program D. If P is feasible, then D is not unbounded, since we should have an upper bound on D. Similarly, if D is feasible, then P is not unbounded, since we should have a lower bound on P.

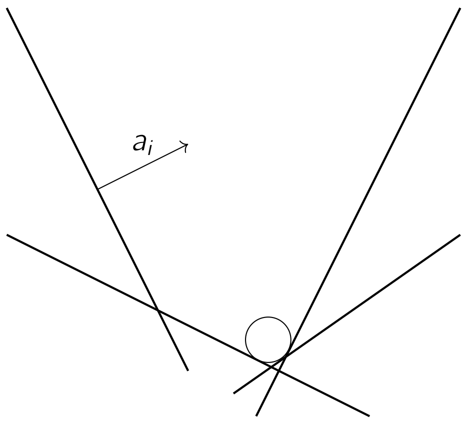
If P is feasible unbounded, then D has to be infeasible, since we can get arbitrarily large v^*. Similarly, if D is feasible unbounded, then P has to be infeasible, since we can get arbitrarily small w^*.

To show that the feasible-bounded but infeasible dual case is impossible, we need to first prove strong duality, seen below. □

With dual programs, we can find an upper and lower bounds on a linear program. How good are these bounds? They're actually strict bounds:

Theorem 15.7 (Strong Duality)
Let v^* be the optimum for a linear program in standard form, and let w^* be the optimum for the dual program. Then $v^* = w^*$.

We'll prove this formally next lecture, but let's see a proof from physics for intuition, since Prof. Karger actually started out as a physics major.



The linear program that we discuss here is to minimize $b^\intercal y$ subject to $Ay \geq c$. We have many hyperplanes, drawn above as lines, which define the polytope of feasibility for y, for which any solution above these hyperplanes will satisfy our LP. If b is pointing straight up, then our optimum is at the absolute bottom of this polytope, which we can find just by dropping a ball into this well and letting gravity act on it. By changing the coordinate system such that b is pointing directly up, this generalizes to any linear program. On this ball, the only forces acting on it are gravity itself and the normal forces exerted by the hyperplanes. The net force on the ball is zero.

The normal forces point in the direction a_i that is normal to each of the hyperplanes (only one is shown above). If the magnitude of each force from wall i is x_i, the condition that the net force is zero implies that the sum of all forces $\sum a_i x_i = Ax = F_g = b$, which is our dual constraint. Further, the magnitudes x_i of the normal forces are all positive, and so the dual has seemingly popped out of nowhere!

We have that $x_i > 0$ only if the ball is touching that wall, since a ball not touching any hyperplane cannot receive any normal force. The magnitudes of the x_i tell us how important each of the constraints are towards our solution, which reflects a type of complementary slackness. Further, $x_i > 0$ implies that the constraint is tight, and hence $a_i y = c_i$.

A compact way of writing these physical observations is then $(c_i - a_i y) \cdot x_i = 0$.This is equivalent since it implies that either $x_i = 0$, for which the hyperplanes exert no force, or that $c_i - a_i y = 0$, meaning that the ball is touching that hyperplane. Summing these constraints over i yields the equality that $c^\intercal x = y^\intercal Ax$, and we know that $Ax = b$ from equilibrium. Thus, at this point $c^\intercal x = y^\intercal b$, and hence we have used physics to show strong duality.

Remark 15.8. *It is not exactly clear here what physical correspondence the objective function of the dual has.*

16 Lecture 16: Linear Programming IV

16.1 Strong Duality

Let's now formalize our physics intuition from last time with some math. Consider the optimum y^* for the dual programs of minimizing $y^\intercal b$ where $y^\intercal A \geq c^\intercal$, and where we want to maximize $c^\intercal x$ subject to $Ax = b$ and $x \geq 0$. We will consider the subset S of maximially linearly independent columns which are tight constraints of A, which helps simplify the proof. We will use the matrix A_s and vector c_s, which are respectively the restrictions of A and c to the subset S. We also have $|S| \leq m$ where m is the dimension of y.

Lemma 16.1

If we can prove strong duality with respect to A_s, then we will also prove strong duality with respect to the original matrix A.

Proof. If strong duality is shown for A_s, then we have some x_s^* such that $A_s x_s^* = b$, satisfying $x_s^* \geq 0$ and $c_s^\intercal x_s^* = y^\intercal b$. Then, if we let x^* be equal to x_s^* with all other entries zero, then this x^* satisfies the original linear program. □

Hence we can just concern ourselves with full-rank matricies, and so we will just assume that A is full-rank now, and drop the subscript s.

Lemma 16.2

If y^* is optimal, then there is some x^* that satisfies $Ax^* = b$.

Proof. Assume by contradiction that there is no x^* such that $Ax^* = b$ (from the physics intuition, this means that the forces are never balanced). Then, from duality of linear equalities (mentioned in the proof of weak duality before), there is some z such that $z^\intercal A = 0$ and $z^\intercal b \neq 0$, which we can assume is less than zero.

Consider $y' = y^* + z$. Then, y' is also a feasible solution, since $y'A = y^*A$. But $y'b = y^*b + zb < y^*b$ contradicting the optimality of y^*. Thus, no z can exist, and hence there is some x^* such that $Ax^* = b$. In the physical sense, if the forces are not balanced, then the ball can move to a better place. □

Finally, we need to show that $x \geq 0$. From a physical standpoint, a negative x means that the normal force is pulling the ball towards the wall, which is infeasible.

Lemma 16.3

A solution must have $x \geq 0$.

Proof. Assume the contrary. Then, there is some $x_i < 0$. Let $c' = c + e_i$, and consider the solution to $y'A = c'$. Since $c' \geq c$ then $y'A = c' \geq c$, and so y' is feasible for the original linear program. The value is $y'b = c'x^* = cx^* + e_i x^* = y^* + e_i x^*$. This means that y' would be a new minimum, which is a contradiction as we assumed that y^* was optimal. □

We can now prove strong duality:

Theorem 16.4 (Strong Duality)

There are values of x and y such that $y^\intercal b = c^\intercal x$.

Proof. The program $Ax = b$ has a feasible solution by Lemma 16.3, and y is optimal by Lemma 16.2. We then have $Ax = b$ and $y^\intercal A = c^\intercal$. Then, multiplying the first equation by $y^\intercal$ yields the theorem, and by Lemma 16.1 this is true for all linear programs. □

From strong duality, we can actually conclude a unintuitive fact:

Corollary 16.5

Optimization of a linear program is as hard as finding a feasible point in the linear program.

Proof. Consider where we have a primal program to minimize $c^\intercal x$ subject to $Ax = b$ and $x \geq 0$, and also to maximize $y^\intercal b$ subject to $y^\intercal A \leq c$. Solve both of these simultaneously. If we do, then any feasible solution to the combined linear program corresponds to a feasible optimum in the set of dual programs. Thus, a feasible point finder can act as a optimal solution finder. □

16.2 Taking a Dual

We have shown that duality holds for any standard primal program with its canonical dual. Though we can convert any linear program to these forms, this is a hassle.

Intuitively, adding a primal constraint means adding a dual variable, and making the primal harder makes the dual easier. If the constraints are as tight as possible (equality) then the dual is unrestricted in sign. A natural constraint is a nonnegative variable. From the physics standpoint, if we had lower bounds on the minimum, then we must have non-negative forces x_i, and likewise for the reverse case.

We can actually dualize every linear program, with a few special rules. Let's look at an example that shows everything we need to take care of:

Example 16.6

Suppose that our linear program is to minimize $c_1x_1 + c_2x_2 + c_3x_3$ subject to $x_1 \geq 0, x_2 \leq 0$, and x_3 unrestricted. The constraints will be as follows:

$$A_{11}x_1 + A_{12}x_2 + A_{13}x_3 = b_1$$
$$A_{21}x_1 + A_{22}x_2 + A_{23}x_3 \geq b_2$$
$$A_{31}x_1 + A_{32}x_2 + A_{33}x_3 \leq b_3$$

For the dual, we want to maximize the quantity $y_1b_1+y_2b_2+y_3b_3$. The constraints then become

$$y_1A_{11} + y_2A_{21} + y_3A_{31} \leq c_1$$
$$y_1A_{12} + y_2A_{22} + y_3A_{32} \geq c_2$$
$$y_1A_{13} + y_2A_{23} + y_3A_{33} = c_3$$

where y_1 is unrestricted, $y_2 \geq 0$ and $y_3 \leq 0$, which comes from the requirement of weak duality that the y serves as a lower bound.

The changes that we have to make are summarized below:

Primal:	Minimization	Dual:	Maximization
Constraints:	$\geq b_i$	Variables:	≥ 0
	$\leq b_i$		≤ 0
	$= b_i$		unrestricted
Variables:	≥ 0	Constraints:	$\leq c_j$
	≤ 0		$\geq c_j$
	unrestricted		$= c_j$

16.3 Duality - Shortest Path Problem

While solving the dual program is mathematically equivalent to solving the primal, frequently, it makes sense to consider the dual, and ask what it means, in order to learn more about the problem itself.

Consider the shortest path problem, where we try to find the shortest path from s to t. If we use a physical interpretation, connecting each vertex with a piece of string, and lifting s, then the tension of each string pulls the vertices up. Some strings will be tight and actually cause a tension, while other strings will be slack and not do anything. The condition for all strings is that the distance between two vertices must be less than c_{ij}, since strings are inextendible. The shortest path occurs when there is a set of strings connected from s to t all in tension.

For the shortest path problem, if we let d_v be the height from the ceiling for vertex v, then we then want to maximize $d_t - d_s$ such that $d_j - d_i \leq c_{ij}$ for all i, j. This is our dual linear program we want to work with, where we have n^2 constraints corresponding to the n^2 verticies, and the n verticies corresponding to d_k.

We can write this program in matrix from as $A \cdot \vec{d} \leq \vec{c}$, where A is a $n^2 \times n$ matrix whose rows correspond to each edge. Specifically, for each row a_{ij}, the ith entry in the row is equal to 1 and the jth entry is equal to -1, with all entries corresponding to 0. If there is no edge between two vertices, we will set $c_{ij} = 0$.

For the dual, we will have n^2 variables corresponding to our n^2 original constraints. We have one variable y_{ij} for every edge, and our goal will be to minimize $\sum c_{ij} y_{ij}$. Our n constraints will correspond to each vertex v: $\sum_j y_{ji} - y_{ij} = 0$ if $v \neq s, t$, while $\sum_j y_{js} - y_{sj} = -1$, and $\sum_j y_{jt} - y_{tj} = -1$, where $y \geq 0$. This is a min-cost flow problem of sending one unit of flow!

This is amazing. Just by taking the dual, we have already found an alternative formulation of the shortest paths problem, through pure algebra.

17 Lecture 17: Linear Programming V

We saw that mechanically taking duals are very helpful in providing insight into some problems. Let's now look at the max flow problem, framed in terms of a linear program:

17.1 Duality - Max Flow Problem

We will first change this into a circulation by adding a $t \to s$ edge, making the notation simpler.

Suppose we have x_{vw} flow from v to w. Then our objective is to maximize x_{ts}, and our constraints are the conservation, capacity, and positivity constraints:

$$\forall v : \sum_w x_{vw} - x_{wv} = 0$$

$$\forall v, w : 0 \leq x_{vw} \leq u_{vw}$$

Since we are trying to maximize x_{vw}, this is the dual, and we need to find the primal as a minimization problem.

We have $n + m$ constraints, corresponding to the number of vertices and edges (with $u_{vw} = 0$ if there is no edge). When we take the dual, we will have $n + m$ variables, let us say, z_v, y_{vw}. Then, the objective becomes minimizing $\sum u_{vw} y_{vw}$.

We had m variables earlier, and so we will now have m constraints. If we consider the matrix representation of the dual program, then we can simply look at the columns to find the constraints necessary for the primal. The matrix for the dual program is a $m + n \times m$ matrix, with an identity matrix of dimension m on the top, and then n rows with column entries equal to $1, 0$, or -1 depending on whether an edge enters or exits that vertex.

Let's now look at the columns, so we can take the dual. In a specific column in the rows corresponding to the identity matrix, we will simply have a 1 in the row corresponding to y_{vw}. In the vertex rows, we will see a $+1$ if there is an edge corresponding to row z_v and a -1 corresponding to row z_w. Thus, the constraints in the dual simply simplify to $y_{vw} + z_v - z_w$. This value, for each edge, must be ≥ 0 by our results about the dual, and ≥ 1 if $vw = st$. The constraints on sign result in $y_{vw} \geq 0$ and z_v are unrestricted.

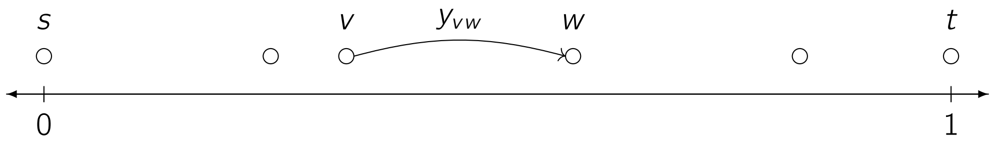
Summing up, our final result is to minimize $\sum u_{vw} y_{vw}$, subject to the constraints that

$$y_{vw} + z_v - z_w \geq f(vw)$$

where $f(vw) = 1$ if $vw = ts$ and 0 otherwise, and $y_{vw} \geq 0$.

Let's now see how we can interpret this new problem. The constraints remind us of the triangle inequality, and we can interpret the y_{vw} as lengths and z_v as distances. We can further assume that $y_{ts} = 0$, since in a standard max flow problem we have $u_{ts} = \infty$. So, our constraint on the edge ts just becomes $z_t - z_s \geq 1$, or that the distance between t and s is ≥ 1. We can further shift all the z_i by z_s, such that $z_s = 0$, since nothing changes in the objective.

Thinking about this problem geometrically now, we have z_s at a distance of 0, and z_t at a distance of at least 1, on a number line. The vertices z_v are distributed along this line between $(0, 1)$ and the length between v and w, y_{vw}, has to be greater than the difference in distances. This is an **embedding** of the graph on a line (example shown just for one pair of vertices):



We needed to minimize $\sum u_{vw}y_{vw}$. What are good edge lengths to use for the y_{vw}? Well, we can interpret the z_v as pipes, and with u_{vw} as a cross sectional area between edges vw. Thus, if we choose good lengths for y_{vw}, then the objective is simply to minimize the total volume $\sum u_{vw}y_{vw}$..

What does this have to do with the max flow problem that we originally started with? Well, if we wanted to send f units of flow from s to t, then we would have needed to send f units of flow across our network, so our volume is at least f. The minimum volume of the network embedding is therefore greater than the volume of flow in the max flow network, providing a physical argument for weak duality! And by strong duality, we know that the minimum volume of the network embedding is actually sufficient to send that required flow.

To proceed, let's take a brief detour and talk about complementary slackness, which will help us complete this analysis.

17.2 Complementary Slackness

We've introduced this term quite a few times before, but let's now introduce this formally for an arbitrary linear program.

Suppose that we have a pair of linear programs, and we have a feasible x and y for respectively the primal and dual linear programs. We define the **duality gap** to be equal to $c^\intercal x - y^\intercal b$, and by weak duality this is ≥ 0. Strong duality tells us there exists x, y such that the duality gap is actually equal to zero.

We will write our dual program as one where we want to maximize $y^\intercal b$ given that $y^\intercal A \leq c^\intercal$, and the primal program as one where we want to minimize $c^\intercal x$ subject to $Ax = b$, with $x \geq 0$. If we introduce a slack variable $s \geq 0$ such that $y^\intercal A + s^\intercal = c^\intercal$, where $s \geq 0$, then we can compute the duality gap as follows:

$$c^\intercal x - y^\intercal b = (y^\intercal A + s^\intercal)x - y^\intercal b = s^\intercal x$$

By weak duality, we know that $s^\intercal x \geq 0$, and this is only equal to 0 (strong duality) if either the $s_i = 0$ or $x_i = 0$. Thus, either x_i is tight at 0, or $y^\intercal A_i$ is tight at c_i. This is the **complementary slackness condition**.

Max Flow Continued

Let's apply complementary slackness to our problem. Consider the optimum of the primal program, where we have optimal y^*, z^*. If we want the minimum volume overall, we should take $y_{vw} = z_w - z_v$, in order to make all those constraints tight, and leading to the smallest objective.

To find the optimal z^*, let's consider the set S consisting of all vertices v such that $z_v^* < 1$. This makes a $s - t$ cut. Note that $z_s \in S$ and $z_t \in \overline{S}$. If an edge vw leaves S, then $y_{vw} \geq z_w - z_v > 0$. Since this is not tight, then by complementary slackness, the corresponding constraint in the dual program must be tight. Thus, $x_{vw} = u_{vw}$ if there is an edge that goes from S to $\overline{S}$.

Suppose now that edge vw enters S. Then, $z_v > z_w$ and since $y_{vw} \geq 0$, we know that $z_v + y_{vw} > z_w$, meaning that the corresponding constraint is slack. This implies that $x_{vw} = 0$. This means that this is a min-cut, and is equal to the value of the flow! Every edge leaving the cut is saturated, and every edge entering the cut is equal to 0.

We've just proven the max-flow min-cut theorem without an ounce of combinatorics.

To wrap up solving this program, we can just set $z_v \in S = 0$ and $z_v \in \overline{S} = 1$, and set $y_{vw} = 1$ for cut edges. This satisfies all the constraints of the primal program, and corresponds to a max flow in the original graph. Complementary slackness showed that we can have a solution which corresponds to a cut in the graph, hence showing the max-flow min-cut theorem. (Strong duality could not have done so by itself, since some y_{vw} could have been fractional without much meaning in the original graph).

17.3 Duality - Min-Cost Circulation

The linear program corresponding to the min-cost circulation problem is very similar to that of the max flow. The only difference is in the objective function, where here we want to minimize $\sum c_{vw} x_{vw}$. The constraints remain the same.

For the dual, everything likewise remains the same, except for the fact that we need to maximize the objective $\sum u_{vw} y_{vw}$, and the variables on the right hand side of the inequalities also change. The constraints then become $y_{vw} + z_v - z_w \leq c_e$ and $y \leq 0$. If we rewrite $p_v = -z_v$, then our condition is that $y_{vw} \leq c_{vw} + p_v - p_w$, which is equivalent to $y_{vw} \leq$ the reduced cost!

Now let's apply complementary slackness. If $x_{vw} < u_{vw}$ then the constraint is slack, and in the optimum the reduced cost is $y_{vw} = 0$. If the reduced cost is less than 0, then $y_{vw} < 0$ is slack, and hence $x_{vw} = u_{vw}$ - the edge is saturated. If the reduced cost is greater than 0, then $x_{vw} = 0$, i.e. there is no flow along that edge. With linear programming techniques, we have thus recovered our previous results of min-cost flows as well.

18 Lecture 18: Linear Programming VI

Now that we've talked a lot about the structure of linear programs, let's talk about how we can solve them. Earlier we already found an algorithm that works by simply trying every basic feasible solution, but this was too slow.

18.1 More on Basic Feasible Solutions

To improve, let's be smarter about our search. We'll work with linear programs in standard form, $Ax = b, x \geq 0$. We can assume that A is full row rank, since all the other rows of linear combinations of constraints are redundant (or contradictiory).

Suppose that the matrix A contains m constraints (or that A is $m \times n$). Then, at a basic feasible solution, all the $Ax = b$ constraints are tight, and we must have $n - m$ tight constraints with respect to the $x \geq 0$ constraints, which means that $n - m$ of the variables must be equal to zero. If we rearrange and write out our matrix equation, we get the following:

$$\begin{pmatrix} A_1 & A_2 \\ I_{n-m} & 0 \\ 0 & I_m \end{pmatrix} \cdot \begin{pmatrix} \\ x \\ \\ \end{pmatrix} \begin{matrix} = \\ = \\ \geq \end{matrix} \begin{pmatrix} b \\ 0 \\ 0 \end{pmatrix}$$

where $A = (A_1 | A_2)$.

In the top n rows (consisting of the matrix A, I_{n-m}, and the zero matrix next to it), we have a basis by definition, and so those rows are linearly independent. Similarly, the columns of that square matrix must also be linearly independent. It is clear that the first $n - m$ columns are linearly independent, and the last m columns must also be independent for a solution to be a basic feasible solution.

Thus, we arrive at another definition of a basic feasible solution: m linearly independent columns of A constraints cover all slack variables $x_j > 0$. The set of the last m columns is called a **basis** (it is a mathematical basis, but this term has a more specific meaning in the simplex algorithm). The corresponding nonzero x_j are called **basic** and the other variables are called **nonbasic**. In fact, if we have the set B of basic variables, then we can compute x. We can do this by considering A restricted to the columns corresponding to B (a full rank matrix having dimension $m \times m$,) and then inverting this matrix A_B to recover x (the other $x \notin B$ are equal to zero).

Note that we can have multiple bases B for the same vertex x, since we can have arbitrarily many tight constraints at zero. Only when we have m slack variables can we have a unique basis.

In summary, x is a vertex if there is a basis B such that A_B is $m \times m$ nonsingular, where $x_B = A_B^{-1}b \geq 0$, and all other $x_i = 0$.

18.2 Simplex Method

The simplex method starts at a basic feasible solution, and repeatedly moves to a better one, via local search. The idea is to follow an edge to a new vertex, and continue repeating this until we get to the optimum. Since an edge is defined by $n-1$ tight constraints, we are essentially moving along this edge by swapping out one of the tight constraints for another.

Let's look at the mathematical details now. We break up A as $\begin{pmatrix} A_N & A_B \end{pmatrix}$ and x as $\begin{pmatrix} x_N & x_B \end{pmatrix}^T$. Then, the condition that $Ax = b$ is equivalent to $A_N x_N + A_B x_B = 0$, which is true for all feasible x (not just where $x_N = 0$, which happens at a vertex). Then, we can solve for $x_B : x_B = A_B^{-1}(b - A_N x_N)$. We can also write the objective as $c^\intercal x = c_B^\intercal x_B + c_N^\intercal x_N = c_B^\intercal A_B^{-1} b + (c_N^\intercal - c_B^\intercal A_B^{-1} A_N) x_N$. The first quantity is constant for a specified basis, and we call the second our **reduced cost** $\tilde{c_N} = c_N^\intercal - c_B^\intercal A_B^{-1} A_N$. Then, our condition for optimality occurs when all components of $\tilde{c_N} \geq 0$, since no feasible points have a smaller objective.

If some nonbasic $c_j < 0$, this means that we are not at an optimum. We can improve by increasing x_j, since when x_j gets bigger, the objective decreases. (Note: this also modifies x_B) We continue until we hit $x_i = 0$ for some basic $x_i \in x_B$. Then, we can swap x_i and x_j - x_j becomes slack and enters the basis, while x_i becomes tight. The process of swapping these basic variables is known as a **pivot**. Thus, a simple way to describe the simplex algorithm is to pivot until the optimum is found.

While this seems like a good algorithm, we have a few problems. One problem is that we can have cycles of pivots. This is since some x_i may already be at 0, and increasing a nonbasic x_j would lead to infeasibility. Though we can swap x_i and x_j, this doesn't change the objective, and hence there is a possibility of cycling pivots. In order to avoid this, we need to design a guaranteed noncycling pivot rule. Indeed this is possible, with the "lexicographically first" heuristic. This works since you're always improving the lexicographical order of the basis, and so you never get the same basis more than once. Thus, we have a finite runtime (though of runtime proportional to $\binom{m}{n}$)

Another problem is the finding of a feasible point that we need at the start of the algorithm. We know that finding a feasible point is very hard in general, but not for specific polytopes. We can do so by optimizing a different linear program Q, for which finding a feasible point is easy. The optimum of Q then gives a feasible solution for the original LP (this will be fleshed out on a PSET). With these changes, the runtime is still bad, and is not apparently polynomial. In practice, however, simplex is usually fantastic, fast, powerful, useful, and easy.

Over the last 60 years, people have been trying to prove if there are pivoting rules that result in polynomial runtime or not. Klee and Minty made the Klee-Minty Cube, which is a twisted hypercube, for which many pivoting rules visit all vertices on this cube (leading to an exponential lower bound for these pivoting rules). Hirsch made a conjecture that if there is a polytope, that the maximum number of pivots required, equal to the diameter of the polytope, is $m+n$, but this was disproved by Santos, who showed that the diameter is $\geq (1+\epsilon)(m+n)$. This is still good enough for our purposes, but the pivot path may not be monotonic, as the simplex requires. Kaloi and Kleitmen showed that the path length is at most $m^{\log n}$ and also gave a simplex algorithm with a runtime of $2^{\sqrt{n}}$. Spielman and Teng used smoothed analysis to show that any linear program, plus small random perturbations, ensures the existence of a fast simplex algorithm.

18.3 Simplex and Duality

Recall from earlier that we defined the reduced costs $\tilde{c_N} = c_N^\intercal - c_B^\intercal A_B^{-1} A_N$ and we have an optimum when all $\tilde{c_N} \geq 0$. If we define $y^\intercal = c_B^\intercal A_B^{-1}$, then $y^\intercal A_N = c_N - \tilde{c_N} \leq c_N$ at an optimum. Then, we have that $y^\intercal A \leq c$. We also have that $y^\intercal b = c_B^\intercal x_B = c^\intercal x$, or that y is a dual optimum! Thus, finding the basis for an optimum x in the simplex algorithm, also finds the optimum y in the dual. More generally, every basis gives some y, and the duality gap $y^\intercal b - c^\intercal x$ always decreases, letting us know exactly how far we are from the optimum. This gives another way to prove strong duality, by showing that simplex terminates (showing that the duality gap goes to zero).

19 Lecture 19: Linear Programming VII

19.1 Ellipsoid Method

The ellipsoid method is essentially a fancy binary search algorithm. Let's first consider a simple problem, which we know how to solve:

Example 19.1

If I'm thinking of a number between 1 and 100, then how do you find what it is? The obvious answer is binary search.

But this is a non-answer for many cases! If we had no bounds on what the number could be, then we can't start the search at all! The second case is the assumption that the 'number' is not necessarily an integer, and where we once again need infinitely many searches. This can be bounded by the bit length of the solution if this is known.

Ellipsoid essentially generalizes this approach to many dimensions. It works as a feasible point algorithm, rather than an optimization problem, but it doesn't matter as discussed before.

Let's now try the above binary search to a polytope that defines a linear program. We can start by splitting the volume of feasibility with hyperplanes, but this requires exponentially many iterations, similar to what we had earlier, since we don't know where the polytope is. How do we find a feasible point? Well, the moniker of the ellipsoid algorithm comes in handy, in where we draw a very large ellipsoid, and then shrink the ellipsoid depending on where the region of feasibility is of a specific hyperplane.

More specifically, we will query the center of the ellipsoid. If the center is in the polytope, then we are done. Otherwise, there should be a separating hyperplane through the center of the ellipsoid that doesn't touch the polytope, which has the polytope entirely onto one side. The reason is that there is some violated constraint (since the center is not feasible), and we can simply shift that constraint to the center of the ellipsoid.

The hard part is actually making the smaller ellipsoid, given this hyperplane. We essentially want to keep all the points that the hyperplane says are good, but with an ellipsoid. Now is a good time to actually define an ellipsoid, before proceeding further:

Definition 19.2

An **ellipsoid** is a sphere stretched on various axes, defined by a center z and a transformation matrix D on the sphere. It contains all the points x such that $(x-z)^\intercal D^{-1}(x-z) \leq 1$. D has to be $BB^\intercal$ for some invertible transformation $x \mapsto Bx + z$. If D is a diagonal matrix, then this just scales the coordinate axes.

Theorem 19.3

Given a unit sphere and a vertical hyperplane, we can create an ellipsoid that only covers the right hand side of the sphere, with a smaller volume than the one we started with.

Proof. It is clear that the center of the new ellipsoid must be shifted somewhere to the right if we want to cover only the right half of the ellipsoid. So, we will shift it by some ϵ to the right. The new coordinates will be $(\epsilon, 0, \ldots, 0)$, and the definition of this new ellipsoid now includes all points that satisfy $d_1^{-1}(x_1-\epsilon)^2 + \sum_{i>1} d_i^{-1}x_i^2 \leq 1$, where we assume the matrix for the new ellipsoid to be diagonal.

What d_i can we use? Well, the volume of the ellipsoid is just the square root of the product of the d_i, and we want this to decrease. We additionally want it to contain the half sphere that we care about. Let's just plug in points, and let them tell us what we need:

If we plug in the point $(1, 0, \ldots, 0)$ then the condition becomes $d_1^{-1}(1-\epsilon)^2 = 1$, so d_1 has to be $(1-\epsilon)^2$ or greater.

If we plug in the point with all zeros except $x_i = 1$, then the condition becomes $\frac{\epsilon^2}{(1-\epsilon)^2} + d_i^{-1} \leq 1$ and so $d_i^{-1} \leq 1 - \frac{\epsilon^2}{(1-\epsilon)^2} \leq 1 - \epsilon^2$.

The volume is then $\sqrt{\prod d_i} \leq \frac{1-\epsilon}{(1-\epsilon^2)^{n/2}}$. If we choose our d_i to be the minimum possible values, and set $\epsilon = \frac{1}{n}$, then the new volume is $\frac{1-1/n}{(1-1/n^2)^{n/2}} \sim \frac{1-1/n}{1-1/2n} \leq 1 - \frac{1}{2n}$. Note that this matches with our half-length reduction in a standard binary search, when $n = 1$. It takes a bit more work to show that all points in the right half of the sphere are also in the new ellipsoid, which will be omitted. □

Corollary 19.4

Given any ellipsoid, we can perform a shrinking and result in a smaller ellipsoid covering one side.

Proof. Consider the linear transformation bringing that ellipsoid to the unit sphere. Perform it, apply the previous theorem to find a smaller ellipsoid, and then invert the transformation, creating a smaller ellipsoid that covers the original requisite half, with a volume reduction of $1 - 1/2n$. □

Now let's look at runtime. We still have two unanswered questions - how big is our starting ellipse, and how much do we need to shrink? While we couldn't solve this in the 1D case, we do have some new information in the n-dimensional case - namely that of the constraints. If our solution is d bits, then we can look at an ellipsoid of size 2^d to for sure contain the solution. But we earlier showed that the number of solution bits is polynomial in the constraint bits, and so we can start with a very large sphere of size $2^{\text{polylog(input bits)}}$. Note that this covers all vertices, but may not include unbounded solutions.

Now, for our ending size, for now, suppose that the solution has full dimension. Then, all the coordinates are polynomial size, and the volume is proportional to the determinant, which is also polynomial size. Thus, the ending volume is $\geq 2^{-\text{polylog(input bits)}}$. We thus need a shrinkage of $4^{\text{polylog(input bits)}}$. $2n$ reductions will shrink by approximately $\frac{1}{e}$, and so we can finish the shrinking in (weakly) polynomially many steps.

With the same argument, we can show that it takes $O(\log \frac{1}{\epsilon})$ steps to get within ϵ of the feasible points. We can use this to get around the volume problem. Specifically, if we know that the input size, then there is a certain minimum distance between lattice points, let us say, d. Now, we look at each constraint, and if we say that the constraint is satisfied within a distance of $\epsilon = \frac{d}{3}$ of the actual constraint, then the volume of feasibility expands to a nonzero amount. The ϵ is chosen small enough such that this coverges uniquely on one lattice point after this many iterations.

Remark 19.5. *You need a book to fill in the handwavy details of this description. Recommendation: Grotschel, Lovasz, Schrijver - a seminal work that does everything here carefully.*

Note 19.6

While we could just do the method of feasibility-optimality interconversion with attaching the primal to the dual, what we can instead do is binary search on the optimum. Specifically, add a constraint that says that the optimum is greater than some quantity, and then just binary search until you get feasibility.

Consider the **separation problem** the problem of finding a violated constraint, given any infeasible point. The ellipsoid algorithm shows that from a separation oracle, we can find feasibility, and hence find optima, in polynomial time. This is useful when we have exponentially many constraints, but a short description of the solution. This shows the counterintutive fact that after examining polynomially many constraints, we can actually satisfy all exponentially many. We can further show that separation is equivalent to optimization (check the referenced book above).

19.2 Interior Point Method

The ellipsoid algorithm is horrible in practice - even in the best implementations, the runtime is about $O(n^6)$. The **interior point method**, published in 1985 by Karmaker, rather builds on the simplex algorithm. Instead of going along the boundaries of the polytope as in the simplex, the interior point method, as suggested by its moniker, goes through the interior of the polytope.

This avoids the complex boundary structure that we had with the simplex. If we consider the physical intuition from before, the worry about the traditional simplex algorithm is that the ball will roll down, hit all the walls, and bounce around. What we can introduce is a magnet next to each wall, that basically repels the ball, and lets it roll smoothly without touching walls.

Mathematically, this is represented as a potential function, but here the potential function is used in its actual implementation, rather than just an analysis tool.

When we have a linear program $Ax = b$ such that $x \geq 0$ and we are trying to minimize $c^\intercal x$, then solving the first constraint is easy with linear algebra, but the second constraint is hard to satisfy due to all the hard corners that it makes. Thus, we can define a **logarithmic barrier function** $G(x) = c^\intercal - \mu \sum \ln x_i$, and minimize that instead. If any of the x_i are close to 0, then this barrier function is large, and if x_i is negative, then the function is infinite. This is the mathematical implementation of our magnets.

To actually perform the algorithm, we start with a large positive x, and head towards the minimum of $G(x)$. We do so using gradient descent, where we project the gradient onto the $Ax = b$ subspace (to maintain feasibility). This will steadily head towards the minimum, without letting any of the x_i go to zero.

There is a lot of mathematical machinery that we use to prove these bounds, and there are also some issues that arise if the gradient is perpendicular to the subspace. If we ignore this, we also have the problem that we don't actually find the minimum, especially if the x_i are close to 0. To get around this, we can make μ vary with x_i, and we can use rounding to find the actual lattice point solution. This is also weakly polynomial, but much more practical, and sometimes even beats the simplex algorithm.

Now, this is the state of the art of linear programming techniques. There is one central question remaining:

Question 19.7. *Is there a strongly polynomial linear programming algorithm?*

We don't know. But if you answer this question, then you get an A, and can read many interesting papers along the way.

20 Lecture 20: Approximation Algorithms I

Up until now, we've been talking about **tractable** problems, which we could solve in polynomial time and the central question was how fast we can solve them. Now, we're going to switch tackling **intractable** problems, where we now care about how good we can solve the problem. While most of these intractable problems are known to be NP-complete, some other problems can be solved in polynomial time, but are way too slow.

Alternatives to complete solutions can include solving special cases in polynomial time, showing that things are polynomial for random inputs, or trying to improve the runtime of an exponential algorithm to a smaller exponent. Another method is to use a heuristic, that is generally fast in practice.

20.1 Introduction to Approximation Algorithms

Another way to do so is through **approximation algorithms**, which we will talk about for the next few lectures. We'll be talking about **optimization problems**, in where we have instances I, solutions $S(I)$, and a value function $f : S(I) \to \mathbb{R}$. The goal is to maximize or minimize f, and find the best solution $OPT(I)$.

Example 20.1
One problem we'll be taking a look at is known as the **bin packing problem**. We have items of various sizes, and we want to distribute them into bins of size 1, minimizing the size of our bins. An instance is defined by the items that we have, and a solution is a distribution of items to bins. The value function is equal to the number of bins in the solution, and our goal is minimization.

When we solve problems like this, we make some technical assumptions. The inputs and range of f can be assumed to be either rationals or integers, since we don't know how to deal with real numbers. We additionally assume that the value function gives us a polynomial-size number, so we can actually write down the solution. Additionally, we want the problem to be solved in polynomial time, as usual.

Note 20.2
NP-hardness actually concern decision problems, rather than optimization. They ask whether we can decide if some optimum $\geq k$ or not. If we can solve the optimization problem, then we can also solve the decision problem. Thus, if the corresponding decision problem is NP-hard or NP-complete, then the optimization problem is as well.

We can now define an approximation algorithm to be one such that for any input I, outputs a feasible output in $S(I)$.

For the bin-packing problem, one approximation algorithm is simply to put every item into its own bin. Obviously, this is not a very good approximation algorithm, and so we need to somehow define the quality of an approximation algorithm. We can either care about the quality on a specific instance, or the worst case over all instances (what we will mainly focus on).

Our first measure of quality is an **Absolute Approximation**, where we say that an algorithm A is a k-absolute approximate algorithm if for all instances I, we have that $|A(I) - OPT(I)| \leq k$.

Example 20.3

Consider the **graph coloring problem**: Given a graph G, we assign colors to each vertex, such that no neighbors have the same color, find the minimum number of colors necessary. This is a very hard problem and is also very hard to approximate.

There is a easier problem regarding one subset of the general graph coloring problem - **planar graph coloring**. Here, we draw without crossing edges, and the very famous **four-color theorem** says that any planar graph is 4-colorable.

We can easily create a 4-absolute approximation algorithm by simply implementing the proof of the 4-color theorem. But we can do better, by looking at specific cases when graphs are not colorable. 0-colorable graphs are those which have no vertices. 1-colorable graphs are those which have no edges. 2-colorable graphs are those that are bipartite, and we can color these easily with a greedy algorithm. We can therefore be exact on $0-2$ colorable graphs, and so our approximation algorithm is actually a 1-absolute approximation algorithm. We can't actually make any progress on 3-colorability (this can be shown to be NP-hard). This can be made into a 0.5-absolute approximation if we only care about the number of colors we need, by simply outputting that we need 3.5 colors (but we usually care about an actual solution).

There is also a similar **edge coloring problem**, where we try to color a graph's edges. **Vizing's Theorem** states that every graph of degree Δ can be edge-colored with either Δ or $\Delta + 1$ colors, and so we can also get a 1-absolute approximation algorithm for the edge-coloring graph.

However, absolute approximations are often impossible to get exactly:

Example 20.4

Consider the **knapsack** problem, where you are a shoplifter with a sack of size B, and the store has items with profits p_i and s_i. The goal is to find a maximum profit set of items of size $\leq B$. This is an NP-hard problem. We can assume integrality, as usual.

There is no k-absolute approximation algorithm for the knapsack problem, unless $P = NP$. We can show this by multiplying all the prices by $k+1$. Then, the optimum is also multiplied by $k+1$, and in fact all solutions are multiples of $k+1$. If we have a k-absolute approximation algorithm, then we must be within k of the optimal, and so we've found the optimal solution in polynomial time.

Another example where absolute approximations fail is in the maximum independent set or the max clique problem:

Example 20.5

The **maximum independent set** problem is when we want to find a maximum subset of vertices with no connecting edges, in a graph G. This is known to be NP-complete.

There is no k-absolute approximation algorithm for the maximum independent set problem, unless $P = NP$. We can show this by making $k+1$ copies of the graph. Suppose that the optimal value is K. Then, in the new graph, the optimal value is $K(k+1) = Kk + K$. If we find a k-absolute approximation, then we will get a feasible solution with at least $(K-1)k + K$ vertices. The average number of vertices in each subgraph is $K - 1 + \frac{1}{k+1}$, and by the pigeonhole principle, in the new graph there is at least one graph with K vertices, which allows us to find the optimum.

Hence, we instead define a **relative approximation**, in where an α-approximation solution for I has value at most $\alpha OPT(I)$ for a minimization problem, and at least $OPT(I)/\alpha$ for a maximization problem. By definition, $\alpha \geq 1$. An algorithm is called an α-approximation algorithm if it always outputs an α-relative approximation. Since we usually cannot find a absolute-approximation problem, we will focus mostly on relative approximations.

Note 20.6

Some people invert this definition and always have $\alpha \leq 1$. But this should be clear from context.

20.2 Greedy Algorithms

While greedy algorithms are known to give the wrong answer for many problems, sometimes they give a nice approximation.

Example 20.7

Consider the **max-cut problem**, where we want to find a cut with the most crossing edges. The greedy algorithm here just places each vertex on the side which is opposite the most previously placed neighbors. This is a 2-approximation algorithm - the optimal is $\leq m$, and our greedy algorithm can easily be seen to achieve $\frac{m}{2}$, since at least half of the edges enter the cut.

Another problem where greedy algorithms are okay in approximation is the set-covering problem:

Example 20.8

In the **set-cover** problem, we have n items and m subsets, and we want to find a collection of subsets that "cover" the items, or a set of subsets whose union is equal to all items. The greedy algorithm just works by choosing the subset with the most uncovered items.

Suppose that the optimum is k. Then, the first set has to cover $\geq \frac{n}{k}$ items. Then, at most $\left(1 - \frac{1}{k}\right) n$ items remain. After t steps, $\leq \left(1 - \frac{1}{k}\right)^t n$ items remain. When this quantity is less than 1, then we have covered all items. Thus, by solving, we see that $t > \frac{\ln n}{-\ln(1-1/k)} \sim k \ln n$. The greedy algorithm gives us a $O(\log n)$ approximation.

Now, we will tackle the vertex-covering problem, which is similar to the set-covering problem:

Example 20.9

In the **vertex-cover** problem, we are given a graph G, and we want to cover the edges with vertices, or in other words, every edge connects some covered vertex. We can relate this to the set-covering problem by considering each vertex as a set of its incident edges, and this tells us that we have a good $O(\log n)$ approximation algorithm.

We can also relate this to the maximum independent-set problem. The complementary set of vertices is an independent set. Or, in other words, we can partition the graph into a vertex cover and an independent set.

It turns out that we can actually do a lot better than the $O(\log n)$ solution we had before. Suppose that we have an uncovered edge vw. Either v or w is in the optimal solution. If we take both vertices v and w, then we've reduced the size of the optimum by 1 by taking two vertices, and so we can get a 2-approximation algorithm.

Surprisingly, even though the vertex-cover and maximum-independent set problems are complementary, we can't get a good approximation for the maximum-independent set problem! For intuition, consider a problem whose solutions are in $[0, 1]$. If the optimum is close to 1, then 1 is a nice approximation. But the complement therefore has an optimum close to 0, and it is much harder to approximate the complement, since 0 is an ∞ approximation.

20.3 Scheduling Theory

In the scheduling problems, we have a collection of n jobs, which can run on m machines. Our goal is to optimize something, either the total runtime, average completion time, waiting jobs, etc. and we also have constraints. The constraints are on the machines themselves - we may have one machine, parallel machines, related machines and unrelated machines, whose respective problems are labelled as $1, P, Q$, and R. Each machine is allowed to have preemption of jobs, which are represented by release times r_j and d_j (where we can't start before r_j and can't start after d_j). In a general analysis, we will say that we have m machines and n jobs.

Our focus at first will be on the $P||C_{max}$ problem, which tells us that we have identical machines, with no constraints, where we want to minimize the last completion time. In other words, we want to minimize the maximum over all machines of $\sum_{machine} p_i$. This is known to be NP-hard.

Our greedy algorithm will assign a job to the least loaded machine (also known as **Graham's Rule**, showing that being born in the right time gets your name on a simple thing). The average load $\frac{\sum p_i}{m}$ is a lower bound on the optimum, but not good enough for the analysis. For example, if we only have one job, the optimum is m times the average load, and this m-approximation ratio we have shown is unacceptable. We'll finish this problem next time.

21 Lecture 21: Approximation Algorithms II

21.1 Scheduling Theory Continued

We needed to find a good lower bound in order to show our greedy approximation is actually good. Previously, we looked at the average load, but this was a bad lower bound when there are few jobs. Another lower bound is the length of the longest job, but tons of tiny jobs make this a bad lower bound as well. Since they're bad in different situations, let's look at the sum of them instead, which should cover both cases. Let the average load be L and the length of the max job to be p_j.

Consider the max load machine (the one that finishes last). By assumption, we know that when we added this last job, all other machines had a larger load. This means that this machine originally had a lower load than average L' at that time. The finishing time is then less than $L' + OPT \leq 2OPT$.

We can make this bound slightly stronger, since our average at that time does not include the job p_i. We can then say that the finishing time is less than $L' + p_j = L' + \frac{p_j}{m} + p_j\left(1 - \frac{1}{m}\right) \leq \left(2 - \frac{1}{m}\right) OPT$.

Note that this algorithm is online, in that we can start scheduling jobs whenever jobs arrive. Another interesting thing is that this $2 - \frac{1}{m}$ bound is tight. We can show this by starting with tons of tiny jobs ($\frac{m}{\epsilon}$ jobs of size ϵ) and then one big job of size 1. This will relatively evenly distribute the jobs at first, and then add on the big job at the end. An optimal algorithm would be to give the big job to one machine, and then schedule all the small jobs greedily. This actually achieves the upper approximation ratio of $2 - \frac{1}{m}$.

We can get around this issue with a heuristic of scheduling the longest job first, known as the **longest processing time (LPT)** heuristic. It's actually possible to show that this gives a $\frac{4}{3}$-approximation, but is not online.

21.2 Approximation Schemes

Now that we've explored some greedy heuristics, let's consider approximation schemes in general, which answer the question of how good we can get. It's not exactly clear how we can even approach proving this. Using techniques from complexity theory, we can define an **approximation hardness**, which gives some lower bound on how good an approximation algorithm can be, unless $P = NP$.

Definition 21.1

We define an **approximation scheme** for a problem to be a family of algorithms A_ϵ such that A_ϵ is a $(1+\epsilon)$-approximation algorithm that runs in polynomial time.

There are some problems for which we can get arbitrarily good approximations running in polynomial time, for a set epsilon. This last point is important in that, for example, a runtime of $O(n^{1/\epsilon^2})$ would be a feasible approximation scheme. So, we define another class of runtimes, **fully polynomial approximation schemes (FPAS)**, which have runtime polynomial in n and $\frac{1}{\epsilon}$.

Let's first explore FPASs, which are generally easier to think of, but also have a nice connection to psuedopolynomial algorithms. In general, these are the best that we can possibly do, since NP-hardness ensures that no polynomial algorithm exists (unless $P = NP$).

21.2.1 Knapsack Problem Revisited

Consider the knapsack problem again, and try to find a psuedopolynomial algorithm for integer profits (the integer sizes case will be on the homework). We can use dynamic programming to solve the problem. If we define $B(j, p)$ as the smallest set (in size) from items 1 to j that achieves profit p, and compute $B(n, p)$ for every p, then we can solve the problem by simply searching for the largest profit achievable that is under the required size. It is easy to see by casework that $B(j + 1, p) = \min(B(j, p), s_{j+1} + B(j, p - p_{j+1}))$. We have np problems, each of which take $O(1)$ time to compute, resulting in a total runtime of $O(np)$, where p is the max profit overall, or $O(n^2 P)$, where P is the maximum profit of one item. This is psuedopolynomial (not weakly polynomial, since otherwise this would show $P = NP$). Since knapsack is not NP-hard with respect to the input numbers, we call the knapsack problem **weakly NP-hard**. Other problems are **strongly NP-hard**, which says that there is no polynomial time algorithm even if the numbers are small.

We can convert this psuedopolynomial algorithm into a FPAS. To do so, we can scale the numbers down to a reasonable size. Suppose that our optimum is some profit p. Then, if we scale each profit p_j down to $\frac{n}{\epsilon p} p_j$, rounding down as necessary, then the new optimal profit is $\sum \left\lfloor \frac{n}{\epsilon P} p_j \right\rfloor \geq \frac{n}{\epsilon P} \sum p_j - \sum 1 = \frac{n}{\epsilon} - n$. Though the new set of items may be different than the one before, it will achieve this value or higher when we scale back up.

After scaling, the largest p_j is polynomial in n and $\frac{1}{\epsilon}$, specifically with size $O\left(\frac{n}{\epsilon}\right)$. The runtime is then $O\left(\frac{n^2}{\epsilon}\right)$. Scaling back up, we find a solution of value greater than $\frac{\epsilon p}{n}\left(\frac{n}{\epsilon} - n\right) = (1 - \epsilon)p$. This is a FPAS!

However, how do we actually execute this algorithm, if we don't know p beforehand? Well, we know that we just need a lower bound on what P actually is. And we have a lower bound L - the maximum of the p_i. We also have an upper bound on the total profit - the sum of all p_i. Scaling then reduces the new profit to $\frac{n}{\epsilon L} \sum p_i \leq \frac{n^2}{\epsilon}$. This is still psuedopolynomial, but with runtime increased to $O\left(\frac{n^3}{\epsilon}\right)$.

A faster method to do so is to just guess p, and to run the algorithm enough times to get information on whether p was too high or too low. If we guess $p \geq \frac{OPT}{1-\epsilon}$, then OPT scales to a solution of value $\leq \frac{n}{\epsilon} - n$. This will tell us that our guess was too big. On the contrary, if we guess $p < OPT$, then we will scale to a value of at least $\frac{n}{\epsilon} - n$, which may not be the optimum. However, we will find a guess of sufficient value to tell us that our solution was too small. Binary search with the same lower bounds and upper bounds as before (the maximum of all p_i and the sum of all p_i), we can find a solution in time $O\left(\log\left(\frac{1}{\epsilon}\log n\right)\right)$, since we only have $\log_{1+\epsilon} n = \frac{1}{\epsilon}\log n$ possible trials - $min, (1+\epsilon)min, (1+\epsilon)^2 min, \ldots, max$.

This is a very general procedure to turn a psuedopolynomial algorithm into a FPAS. Similarly, we can also go the other way, creating a psuedopolynomial algorithm from a FPAS. If we suppose that the input uses integers bounded by t, then the optimum is at most nt. We can then choose $\epsilon = \frac{1}{1+nt}$ to solve the problem exactly, since we're guarenteed an integer solution. This is polynomial in n and t. This creates an equivalence between psuedopolynomial algorithms, FPASs, and weakly NP-hard problems.

21.2.2 Enumeration Techniques

Now let's look at strongly NP-hard problems. To approximate this, we use a technique known as **enumeration**, which is a method of controlled brute force. Let's consider the scheduling problem $P||C_{max}$ again. We will schedule the largest k jobs optimally by brute force, and then use the greedy algorithm on the remainder.

How bad can things get? As before, we added the max load to a machine with was free the earliest, which was below average load. This means that when we add that job, the machine takes time at most equal to the optimum plus the size of a remaining job. If we consider the machine with the worst completion time, then we have two cases - either this was scheduled by the brute force approach (so this was optimal) or that it's bounded by time $OPT + p_j$, which is a $1 + \frac{p_j}{OPT}$ approximation. The p_j itself is not one of the k largest, and thus is $\leq \frac{m}{k}OPT$. Substitution yields a $1 + \frac{m}{k}$ approximation algorithm. The runtime is $O(m^k n)$ to brute force the first k jobs, and then run the greedy algorithm. If m is constant, then this is polynomial in k, and this is a polynomial approximation scheme (but not fully polynomial). This is not satisfactory, but can be generalized to non-constant m, as we will see next time.

22 Lecture 22: Approximation Algorithms III

22.1 Enumeration Techniques Continued

We can do better with the enumeration technique that we talked about previously, which solved the scheduling problem $P||C_{max}$. Instead of asking to optimize the solution fully, we can instead change it into a decision problem and ask if we can complete the job in time T. We can optimize with a binary search over T, and we can get the bounds of the binary search easily - the lower bound can just be the average load, while the upper bound is the sum of all job times. Their ratio is m.

Now, we can optimize in the same way that we did for knapsack - try the lower bound, then the lower bound times $(1+\epsilon)$, then $(1+\epsilon)^2$, then so on, until we get to the upper bound. We have $\log_{1+\epsilon} m = \frac{1}{\epsilon}\log m$ values and thus it takes $\log \frac{\log m}{\epsilon}$ iterations. We can't decide this problem in polynomial time, but we can give one of time $< (1+\epsilon)T$ if there is a solution that takes time T.

We can now combine enumeration and rounding to solve this problem. We're first going to consider a special case, where we only have k distinct sizes of jobs. Then, even if we are exponential in k, this will still only be polynomial in n, since k is constant. We will define a **machine type** as a vector of the number of jobs of each size, and we will limit the machine type to contain only solutions whose total size is $\leq T$. Thus, only the feasible job schedulings are included in the machine types. Since we have n jobs, there are $\binom{n+k-1}{k-1} = O(n^k)$ possible machine types. There are also an equal number of $O(n^k)$ ways to specify a job input.

Now, we have a nice optimal substructure: when we schedule on m machines, this is equivalent to scheduling on $m-1$ machines plus 1 other machine, and this calls for dynammic programming. Our algorithm will enumerate all inputs that are feasible (in time T) for r machines, where r goes from 1 to m.

When $r=1$, we can simply enumerate all the feasible machine types. When we need to go from $r \to r+1$, then we can simply take any r-feasible input, and add the jobs on some machine type. This takes $O(n^k)\cdot O(n^k) = O(n^{2k})$ time for every iteration of the loop. The overall runtime is then $O(mn^{2k})$, which is polynomial for constant k.

How do we approximate for nonconstant k? Well, we can simply round the job sizes until we get a constant amount. It turns out that rounding to powers of $(1+\epsilon)$ is good, since each job size will increase by at most $(1+\epsilon)$. Unfortunately, this may still result in many job sizes, especially when we have geometrically decreasing sizes.

Remember from before that if we can schedule the large jobs optimally, that the small jobs are okay since we can just run a greedy algorithm. What if we try to round only the big jobs, and leave the small jobs intact? Specifically, we will round all jobs of size $p_j \geq \epsilon T$. Our jobs then range from size ϵT to T, with increments of $(1+\epsilon)$, and so the total number of job sizes is $\log_{1+\epsilon} \frac{1}{\epsilon} = \frac{1}{\epsilon} \log \frac{1}{\epsilon}$.

We can then solve the problem of the big jobs in time $O(n^{2k}) = n^{O(\frac{1}{\epsilon}\log\frac{1}{\epsilon})}$. If a large job finishes at the end, then we are actually within $(1+\epsilon)$ of the original optimum. On the other hand, if a small job finishes at the end, the total time it takes is $\leq (1+\epsilon)T + \epsilon(1+\epsilon)T = (1+\epsilon)^2 T = (1+2\epsilon)T$, with the greedy algorithm. Thus, we can get an ϵ-approximation in $n^{O(\frac{1}{\epsilon}\log\frac{1}{\epsilon})}$.

This wraps up our discussion of approximation schemes, and it has actually been shown with techniques from complexity theory that any approximation scheme can be gotten with enumeration techniques. Thus, it's always a good idea to turn to it when tackling these problems.

22.2 MAX-SNP hard problems

Sometimes, we don't have any approximation scheme. We formalize this as a class of problems in the class **MAX-SNP hardness**, or max-syntatic NP hardness, where we can show that there is some constant factor from which beyond they cannot be approximated. Since we can't beat that particular constant, then there is no PAS possible. Further, for some other problems, one can show that through **amplification** that MAX-SNP hardness implies that some problems have no constant factor approximation.

A celebrated result from the 1990s is that the max-clique problem has no constant factor approximation. This will be on the homework. Modern amplification techniques have become even more powerful, showing that you can't get a $O(n^{1-\epsilon})$ approximation of the clique problem. Other examples of MAX-SNP hard problems are the vertex-cover and set-cover problems.

22.3 Relaxations

Joke 22.1. *This is always a good topic for MIT students, they need to learn more about this.*

The idea of relaxation is to start with a non-solvable problem, change it to a solvable problem which is similar to the original, solve that similar problem, and then round the solution to the original. Essentially, during relaxation, we remove or lessen certain constraints, increasing the feasible solution space. We can find one solution in this space, then move back into the original feasible space by rounding.

Our first example is the **Traveling Salesman Problem**, which is a highly-celebrated problem of which multiple books have been written. The problem involves a graph G, and edge lengths, where our goal is to visit each vertex exactly once and return to the start over a minimum distance. This is strongly NP-hard, and this can be easily shown by reducing from a well-known NP-hard problem of finding Hamiltonian Cycles (paths which visit each vertex once).

We'll concern ourselves with the **Metric TSP Problem**, which says that the lengths satisfy the triangle inequality, or equivalently, we visit each vertex at least once. These definitions are equivalent, since we can create the **metric completetion**, replacing edges with those whose length is the shortest path between those two vertices in the original graph.

We will work with the undirected version by relaxing to a MST. This is since any feasible TSP is a minimum spanning tree, plus one edge! This tells us that the optimum MST costs less than the optimum TSPs, and so we just need to find some way to convert it to a feasible instance. One way to do so is just to do a DFS, or an inorder traversal of vertices, which visits all vertices at least once. Each edge is traversed exactly twice, and so the cost of the tour is at most 2 times the cost of the MST, and so this gives us a 2-approximation.

Remark 22.2. *The metric TSP problem is MAX-SNP hard, even if all the edge lengths are* 1 *or* 2. *It turns out that the lower bound is* $1 + \epsilon$, *which is pretty low.*

We can do better than a 2-approximation, using **Christofides' heuristic**. This also starts from an MST, but performs a smarter rounding approach. In our previous approach, we essentially toured all edges twice, but we can do much better. Note that all the feasible tours will be **Eulerian**, meaning that we will have even degree everywhere, making it possible to actually tour. So, the idea is to add enough edges to make the MST Eulerian, and then use an Euler tour on those edges.

The bad vertices are, almost by definition, the vertices with odd degree. We need to add edges to the odd vertices, and we need to add $\frac{V}{2}$ edges, where V is the number of odd vertices. This turns into a min-cost matching problem. We can find a minimum cost matching by considering the optimal TSP. If we match the odd vertices in the optimal with one immediately clockwise of themselves, we can do so in two ways, and the sum of these costs equals the total cost of the TSP. Thus, one of these matchings is less than $\frac{1}{2}$ of the TSP cost, and so this leads to a $\frac{3}{2}$ approximation.

Remark 22.3. *We didn't actually discuss how to solve min-cost matching, but this is possible in polynomial time, without solving the TSP problem.*

Christofides proposed this algorithm in 1983, but it basically was left untouched in 40 years. The Held-Karp relaxation was also proposed, where no example shows an approximation ratio worse than $\frac{4}{3}$, but no one has been able to prove this bound. Finally, this month (October 2020), Karlin, Klein, and Oveis Gharron showed a $\frac{3}{2} - \epsilon$ approximation, where $\epsilon \geq 10^{-36}$. Directed MTSP has seen more progress to a $O(\frac{\log n}{\log \log n})$ approximation. Euclidean TSP, which only tours lattice points, actually has a PAS, which was shown in the 2000s. However, there are still many open problems!

23 Lecture 23: Approximation Algorithms IV

23.1 Relaxation in Scheduling

Let's now look at relaxation algorithms applied to the scheduling problem, where now we will look at the problem $1||\sum C_j$, essentially trying to minimize the average completion time of each job. One easy heuristic is to schedule the shortest processing time jobs first.

Theorem 23.1
This heuristic of scheduling the shortest-processing time jobs is actually optimal. This obviously implies that this problem is not an NP-hard problem.

Proof. Suppose that we are out of order. Then, there is some job such that p_{j+1} is executed before p_j, where the jobs are in processing time order. If we swap the two jobs, then p_j will finish earlier and p_{j+1} will finish later, but by looking at their respective times, this minimizes the objective. □

Now let's look at an actual NP-hard problem, $1|r_j|\sum C_j$, where we try to minimize the average completion time of each job, but cannot process job j until after r_j time has passed. We can't use our previous algorithm, and the natural heuristic of processing the shortest avaliable job doesn't work out, since we could have a large job that is immediately avalible, followed by many short jobs avaliable after a small time. What we want to do is just quit when these new smaller jobs come in, and this leads to another scheduling problem, known as **preemption**.

In the preemption problem, denoted as $1|r_j, p_{min}|\sum C_j$, we are allowed to stop a job, and then restart it later, with the progress being done being saved. The heuristic is then to do the shortest remaining processing time job first (SRPT), and if this is easy to do, then we can relax our original problem to this one.

Theorem 23.2
SRPT is optimal for solving $1|r_j, p_{min}|\sum C_j$.

Proof. Use a similar exchange argument as above, but use parts of problems instead of problems as a whole. Consider that we have jobs j and k which have both been released at some time T, and suppose that the remaining time on j is greater than the remaning time on k. Then, the scheduling of jobs j and k can take place in multiple parts, and the idea is to swap pieces of j and k such that all of j is scheduled before any of k is scheduled. The completion time of j decreases while the completion time of k is the same, while the other jobs are unchanged, ultimately helping minimize our objective. □

Let's now consider how we can convert from $1|r_j|\sum C_j$ to the one with preemption. We can take the preemptive schedule, and then insert p_j units of extra time wherever job j completes. This stretches out the schedule, and this is feasible, since we can simply run the whole job starting when the last part of a job, with the extra time intervals that help us.

For job j, we have a slowdown of the processing of all the jobs finishing before j in the SRPT, for a time of $\sum p_i \leq C_j$. Then, the actual processing time is C_j and so job j finishes in time $2C_j$. This means that this is a 2-approximation algorithm. In fact, it's known that we can use rounding and enumeration to get a PAS for this problem.

This concludes our discussion of scheduling theory, though there are thousands of scheduling problems still left unsolved. There is an online database that includes all the hardest problems that do have polynomial solutions, and the easiest scheduling problems that don't have a known polynomial solution. So, if anyone wants to solve a problem, go there.

23.2 General Relaxation with Linear Programming

Our previous relaxation techniques with MST and SRPT worked fine, but somewhat non-obvious. Now, we will look at a much more general way to do the relaxation - simply write a integer linear program, solve the related linear program ignoring the integrality constraint, then round. This may not always lead to an optimal solution - it depends on the structure of the problem.

23.2.1 Vertex Cover Problem

Consider the vertex cover problem. We will say that $x_i = 1$ if vertex i is in the vertex-cover, and 0 is it isn't. The associated linear program is to to minimize $\sum x_i$, given that $x_i + x_j \geq 1$ for all edges i, j and where $x_i \in \{0, 1\}$.

Let's see what happens if we solve the associated linear program with $0 \leq x_i \leq 1$, and where we round these variables the usual way - of rounding up if $\geq \frac{1}{2}$, and rounding down otherwise. Clearly, the constraints are still satisfied, and so this gives us a 2-approximation.

We already had a 2-approximation from before, so this may not seem like an important result. However, this technique can be generalized to weighted vertex cover, and in fact gives more information about the structure of the graph itself (while the previous 2-approximation was much more local in nature).

In fact, this problem has been studied so much that the linear program even has a name - the **stable set polytope**. Nemhause-Trotter studied this polytope, and showed that every value is either 0, $\frac{1}{2}$, or 1 at the optimum. Further, all 0s are not in the optimum vertex cover, and all 1s are.

Modern progress regarding the vertex-cover problem has led to some results for the MAX-SNP hardness of the vertex cover - $\frac{7}{6}$ was shown in 2001, $10\sqrt{5}-21$ was shown in 2002, and $\sqrt{2}$ was shown in 2017. There is a conjecture known as the **unique games conjecture** that would imply the approximation hardness is $2-\epsilon$. On the other side, we have a known $2-O\left(\frac{1}{\sqrt{\log n}}\right)$-approximation algorithm (though this is worse than $2-\epsilon$, since that term goes to 0 for large graphs.

23.2.2 Facility Location Problem

Another problem that can be solved is the **facility location problem**, where we open facilities with cost f_i to serve clients j, whose distance to each facility is c_{ij}. The objective is to minimize the sum of opening all the facilities, plus the sum of the distances to the nearest open facilities over all the clients. We can assume the costs to satisfy the triangle inequality.

In our integer linear program, we will define $y_i = 1$ if facility i opened, and 0 otherwise. We will also define x_{ij} to be equal to 1 if client j is assigned to facility i, and 0 otherwise.

The objective is then to minimize $\sum y_i f_i + \sum c_{ij}x_{ij}$, and our constraints are that $\sum x_{ij} \geq 1$ for all j, ensuring that all clients are assigned to a facility, and that $x_{ij} \leq y_i$ for all i, j, showing that clients can only be assigned to open facilities.

We now have an ILP, and let's think about how to round this nicely. We plan to assign client j to some nonzero x_{ij}, and further we want to filter out all the bad values (for example, when x_{ij} is very tiny) that we have, since otherwise when rounding we can get huge cost scaleups.

We can write $C_j = \sum_i x_{ij}c_{ij}$, as the average assignment cost for client j. This is also the amount of cost that is allocated to client j in the LP optimum. However, some of the c_{ij} may be huge and would mess with our approximation, and so we have to invoke the principle that not everything can be above average. Specifically, we claim that at most $\frac{1}{\rho}$ total of the x_{ij} is to facilities of assignment cost $> \rho C_j$, since otherwise the average assignment cost would be greater than C_j. We can then zero out all the x_{ij} with corresponding large c_{ij}, and scale up the rest of the x_{ij} and y_i by a factor of $\frac{1}{1-1/\rho}$, for all j, in order for the LP to still be satisfied. This is no longer an optimal solution to the LP, but now at least we get rid of any of the large costs, and so the assignment of all clients j costs at most $\rho \sum C_j$.

Now, we've filtered all the bad assignments, but we still need to find out where we want to open facilities. If the y_i are small, this means that opening the facility isn't paid for, and so we need to find a cluster of facilities whose total $\sum y_i > 1$ (here, we invoke the triangle inequality to say that the increase in costs is small). If we open the minimum cost facility here, then this is a lower bound on the first part of the LP $\sum f_i y_i$.

Formally, we will choose the client with the minimum C_j, and consider all avaliable facilities (i.e. those not zeroed out). Choose the cheapest one and close the others, and send everyone who has a nonzero x_{ij} to any facility in that cluster to that opened facility. The cost of assigning a client j' to this is $\leq 3\rho C_{j'}$, since it takes $\leq \rho C_{j'}$ to go to a facility that client j' was originally assigned to, and it takes distance at most $2\rho C_j \leq 2\rho C_{j'}$ to go from any facility in this cluster to another. Iterating, we can easily assign all our clients, and the total cost will be $3\rho \sum C_j$

Now, let's take a look at how good our approximation is. Looking at the LP objective, we've increased the first quantity by a factor of $\frac{1}{1-1/\rho}$ when we were scaling up the y_i and multiplied each of the client-sums by a factor of 3ρ when we were doing assignment. If we balance these two quantities, we see that $\rho = \frac{4}{3}$ will lead to a 4-approximation on both components, and thus a 4-approximation overall.

This problem shows that even when converting an ILP into a standard LP, converting backwards and rounding is often nontrivial. We needed an extremely clever rounding scheme here for things to work out.

24 Lecture 24: Approximation Algorithms V, Parameterization

24.1 Max-SAT - Relaxation with Rounding

The **Max-Satisfiability Problem** is the birthplace of the concept NP-hardness, and it involves a set of boolean variables and clauses, which are satisfied either when a specified boolean variable is true or false. To satisfy a formula, you need all the clauses to be true.

Example 24.1

The formula $(x_1 \lor x_2) \land (x_3 \lor \neg x_1)$ is an example of a formula discussed in the problem.

The goal is to find a satisfying assignment of booleans to true/false, making all the clauses true. The Max-SAT problem aims to maximize the number of clauses that are satisfiable, since sometimes the formula may not be solvable.

Consider a random assignment, where we set each variable to true or false, with probability $\frac{1}{2}$. For a specific clause, the probability that it is satisfied is $1 - 2^{-k}$, where k is the number of literals in the clause. This is good when k is large, but not that good when $k = 1$. Even when $k = 1$, however, this is still a **randomized 2-approximation**, since the expected value is at least half of the optimum.

We can do better, with LP-rounding. Let's construct a LP, with variables y_i corresponding to the actual boolean variables, as well as the clauses z_j. Then, we want to maximize $\sum z_j$, and we also need the constraints to hold. If z_j is true, then the clause must be true, and the variables in the clause must sum to ≥ 1, taking care of negations by putting a $1 - y_i$. This can be written as

$$z_j \leq \sum_{C_j^+} y_i + \sum_{C_j^-} (1 - y_i)$$

where C_j^+ is the set of non-negated variables in the clause, and C_j^- is the set of negated variables. We now have an integer linear program if $y_i, z_i \in \{0, 1\}$, and we can relax this to a standard LP by saying $0 \leq y_i, z_i \leq 1$, which we can solve. The fractional optimum must be $\geq$ the actual optimum.

Now we need to find a good way to round our variables. There is actually a straightforward mechanical way to round, and is a good thing to try before coming up with more creative rounding schemes (while the analyses are more complicated). We can treat each of the y_i as the probability that the boolean x_i is true, or in other words, we set x_i to true with probability y_i and false with probability $1 - y_i$.

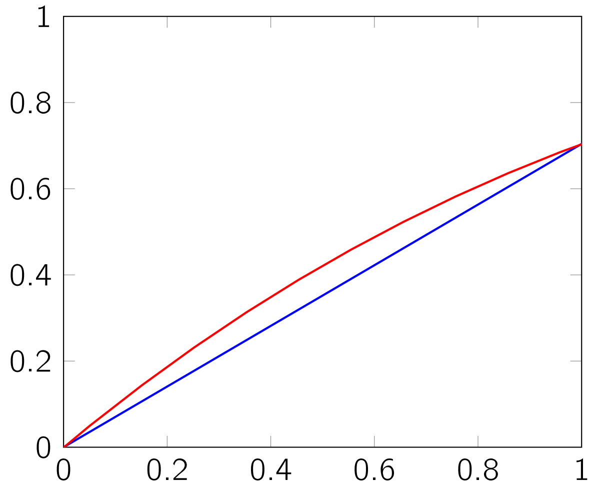
The quantity on the right side of the constraints measures the expected number of true variables in the clause. The expected value of the rounded solution is then equal to the expectation of the number of satisfied clauses, which is the sum of probabilities that each clause is true. We can try to relate z_j to the probability that clause j is true, so that our expectation is related to our objective. Since we want to maximize z_j, if z_j is large, the clause should be satisfied, and similarly, if the expected number of variables on the right that are true is large, then the clause should be satisfied. Let's now formalize this:

Lemma 24.2

Define $\beta_k = 1 - (1 - \frac{1}{k})^k$, which is a decreasing function that converges to $1 - \frac{1}{e}$. Then, if a clause has k literals, then the probability that a clause z_j is satisfied is $\geq \beta_k z_j$.

Proof. First, we can WLOG assume that all variables are unnegated. Then, the probability that the clause is satisfied is 1 minus the probability that it's not satisfied, which is $1-\prod(1-y_i)$. Since we are trying to show a lower bound of the probability of satisfaction, we want to maximize this product. We know that $\sum y_i = z_j$, and applying this constraint, then we want to make all the y_i's equal to maximize. Then, each y_i is equal to $\frac{z_j}{k}$, and the probability of satisfaction is $\geq 1 - \left(1 - \frac{z_j}{k}\right)^k$.

Now we claim that $1 - \left(1 - \frac{z_j}{k}\right)^k \geq \beta_k z_j$, which will show the claim. This is able to be proven pictorally - simply plot both functions as a function of z_j. The function on the right is just a linear function from 0 to β_k, and so is the function on the right. If we consider the second derivative of the function on the left, this is always negative, and hence the function is concave. Thus, it is always $\geq$ the function on the right.



The functions plotted when $k = 3$.

$\square$

Corollary 24.3

This method of random assignment yields a $1 - \frac{1}{e}$ randomized approximation.

Proof. This is trivial from Lemma 24.2. □

Note that this algorithm achieves it's worst behavior when k is large, while our original naive algorithm achieved it's worst behavior when k is small. We can run both, and take the better solution - then this means that the better algorithm will achieve better than the average behavior. The average is

$$\begin{aligned}
&= \frac{1}{2}\left(\sum_j 1 - 2^{-k_j} + \beta_{k_j} z_j\right) \\
&\geq \frac{1}{2}\left(\sum (1 - 2^{-k_j} + \beta_{k_j}) z_j\right) \\
&\geq \frac{1}{2} \cdot \frac{3}{2} \sum z_j
\end{aligned}$$

where we got the constant $\frac{3}{2}$ from the increasing behavior of $1-2^{-k}$ and the decreasing behavior of β_k (minimized average when $k = 2$), and so combining the two strategies we get a $\frac{3}{4}$-randomized approximation. This is actually tight.

Max-SAT is well studied, and we have some pretty tight bounds. We know that it's Max-SNP-hard, meaning that there are no PTAS's, and further, it is known that finding a better approximation than our naive algorithm for Max-3-SAT (where all clauses are 3 variables) is NP-hard.

24.2 Parametrized Complexity

Parametrized Complexity is another way to cope with NP-hard problems. The definition of an NP-hard problem means that the problem is hard over all instances, though this can mean that many instances can be easy! The idea is to define a parameter measuring the hardness of one specific instance, and give a runtime based on that parameter.

Consider the vertex cover problem, and let the parameter k be equal to the optimum. Then, we can find the optimum vertex cover in time $O(mn^k)$, by simply checking all subsets. This is obviously not a very nice polynomial.

A better approach is known as the **bounded search tree** method. We make our choices one at a time, and then consider the solution space as a search tree. We consider our solution space as a search tree, and argue that the tree is shallow if the parameter is small.

For the vertex-cover problem, we can start with one edge, and then branch based on which vertex of that edge we select. When we recurse, we simply remove all the covered vertices. This is a degree 2 search tree (still exponential), and the maximum depth is simply k, since we can find the optimum in k guesses. Thus, if we truncate the search at depth k, the runtime then becomes $O(2^k m)$, since we only need to check at most 2^k guesses. This runtime is a **fixed-parameter tractable** runtime, and in general such runtimes can be written as $f(k) \cdot \text{poly}(m, n)$. Our original algorithm's runtime was obviously not fixed-parameter tractable.

Another way to tackle these problems is known as **kernelization**, which is where we try to find a **kernel** of the problem in polynomial time independent of k. The kernel should be of size $f(k)$, independent of the original problem size. Then, if we solve the kernel in time $g(k)$, we get a fixed-parameter tractable runtime. We can apply this to vertex cover as well.

First, all vertices of degree $> k$ must be in the optimum, since otherwise we must use all it's neighbors, making the vertex cover have size at least $k + 1$. So, we can mark all of these vertices as in the optimum and remove all incident edges. In the new graph, we now only have vertices of degree $\leq k$, and also a vertex cover of size $\leq k$, so only k^2 relevant edges. We can use our previous algorithm on this new graph, which runs in time $2^k \cdot k^2$. Our runtime then becomes $O(m + k^2 \cdot 2^k)$, which is linear time for sufficiently small k.

A third idea to solve problems like this is to consider a **treewidth** of a graph. While the concept of treewidth originally started with theoretical computer science, it's actually now being used widely in machine learning. The idea is to use recursion to solve a graph problem. We pick a vertex, eliminate it from the graph, and then recurse on the remaining graph. However, this may create some 'hidden dependencies' between neighbors, which we can represent by adding more edges in the recursion. If we repeatedly do this, then we get an **elimination ordering** for the graph.

For trees, we can always remove just one leaf, and this basically doesn't affect the graph structure, and so we define the treewidth of trees to be 1. A graph with treewidth 2 is known as a series-parallel graph, where we only have vertices that are connected in series or parallel, allowing us to build up electronic circuits.

There are many problems that are tractable for small treewidth. One such problem is the SAT problem, if we introduce a graph structure such that two vertices will have an edge if they share a clause. We can solve by elimination, with a runtime of $2^{\text{treewidth}}$. To actually implement this, we will take a variable x, look at all the clauses it contains, and find assignments to other variables that also work. This gives a combined clause from all the clauses that contain x, and we can list it's satisfying assignments, henceforth eliminating x. The size of this new clause is equal to the number of neighbors of x, and so is at most the treewidth, and so we obtain the stated runtime.

25 Lecture 25: Computational Geometry I

25.1 Introduction to Computational Geometry

Computational Geometry is its own developed subfield of theoretical computer science, and often focus on low-dimensional objects. It often has runtimes that are exponential in the number of dimensions, but this is okay, since the dimensions are small. We assume that points, lines and planes are the primitives, and that we can check for insersections between primitives, compare lengths of different lines, and compute angles between primitives. Some of these are sketchy since they can involve trig functions and irrational numbers, but we assume that they are tractable primitives and can be $O(1)$ time.

Note 25.1
This is already a simplification from real life, since we have significant roundoff error. In practice, these concerns can actually be resolved, and the algorithms we'll talk about need to have roundoff correction in order to work. But we'll skip that in this course.

For many computational geometry problems, a key idea is to recurse onto a smaller dimension. Since d is already a small constant, then we can do significant work in each recursive step and still end up with a nice runtime.

25.2 Orthogonal Range Queries

Our first problem is known as **Orthogonal Range Queries**, in where we have a set of points as an input, and where we are asked queries about "which points are in this specified box." For the 1-dimensional version, this just asks how many numbers there are in an interval, and this is easily solved with a binary search tree in $O(k + \log n)$ time and $O(n)$ space, where k is the number of numbers in our interval. If we only wanted the number of points, then we just need to store an auxillary variable for each node that tells us the subtree size to compute it in $O(\log n)$ time.

Okay, now let's generalize to a 2D problem - suppose we are now given a bounding box as a query. If we solve each dimension separately, the answer is the intersection of the two 1D answers, which can be very slow. For example, if there are points along the x and y axes, and our bounding box goes from $(1, 1)$ to (n, n), then obviously there are no points in bounding box. However, solving each dimension sepaately takes $O(n)$ time, which is bad compared to the 1D case.

Let's now refine this idea. If we are told what the x-interval of the bounding box is, then we can simply build a BST on all coordinates in the x-interval ordered by their y-coordinate, to solve the problem quickly, as in the 1D case. While there are uncountably many intervals, the only ones we care about are those with endpoints that have different points, and so we can build a BST on each pair of points instead. This gives us a $O(\log n+k)$ query time and $O(\log n)$ count time again, with an additional $O(\log n)$ factor needed for searching for the correct BST. If we generalize to higher dimensions, the runtime will be $O(d\log n+k)$ and $O(d\log n)$ count time, which is fast for small dimension.

The problem is that the preprocessing time and space are large. In the 2D case, we need to make n^2 BST's, each with size n, for $O(n^3)$ space. In the general case, we need to make $d-1$ BSTs for each pair of points, and so this turns into $O(n^{2d-1})$ space, which is very bad.

Instead, let's try to augment each of the BSTs with auxillary information, to help us achieve better queries. We'll build a BST on the x-coordinate, and where each subtree of the BST defines a subinterval of the x-axis. The coordinates themselves will be leaves of the BST, while the internal nodes are empty. The query interval itself is a union of subtree-intervals. Then, our algorithm is just to build a y-BST for the points in each of these x subtree-intervals, and then query these.

Theorem 25.2

For any given interval, there are only $O(\log n)$ subtree-intervals.

Proof. Consider the lowest common ancestor of the left and right coordinates of an interval. If we split the interval at the lowest common ancestor, each side only contains up to one subtree of each height, since containing two subtrees of the same height means that we could have looked at the subtree defined by their parent instead. Since the height is $O(\log n)$, then there are only $O(\log n)$ subtree-intervals. □

Overall, $O(\log^2 n+k)$ time is then necessary for each query, and for general dimension, we need $O(\log^d n+k)$ time. For the preprocessing time, this is simply equal to the time it takes to build the necessary y-BSTs. Since each point is in exactly one y-search tree per level of the x tree, and there are only $O(\log n)$ levels, the total space used is only $O(n\log n)$. When we generalize to d dimensions, this takes $O(n\log^{d-1} n)$ space.

Note 25.3

There is a technique known as **fractional cascading**, where we essentially deal with the other dimensions when we make a higher-dimensional BST, which improves the query time to $O(d\log n)$. We won't have time to go over it in this course, but it should be relatively googlable and understandable with our current knowledge.

25.3 Sweep Algorithms

We talked about **sweep algorithms** previously, where we used persistent data structures to solve a computational geometry problem. The idea is to treat one dimension as the time, and then sweep over the time to solve the $d-1$-dimensional problem at each time. The structure of the problem doesn't change too much over time, and so we can use a solution from a previous time to generate one for the next time.

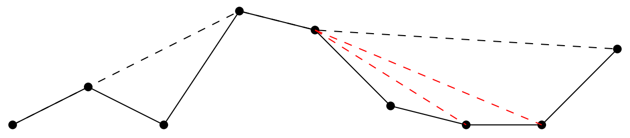
25.3.1 Convex Hull Problem

The (2D) **Convex Hull Problem** involves a given set of points as the input, and as the problem name implies, we want to find the **convex hull**, or the smallest containing convex polygon of the points. One thing to note is that if we can find the points in the convex hull, that we can also find the containing polygon easily simply by connecting these points.

Remark 25.4. *The convex hull problem is to computational geometry as sorting is to general algorithms. There are about 73 different algorithms which each demonstrate a different technique of computational geometry.*

Let's explore the sweep line algorithm for solving the problem. It involves finding the upper hull, which are simply the points of the convex hull that can be seen from above. Then, finding the lower-hull and combining with the upper-hull is enough to find the overall convex hull.

We'll construct the upper-hull first. We'll sort all our points by the x coordinate, which we'll assume is our encounter time. Then, we sweep from left to right, and track what the upper hull looks like "so far." As we encounter a new point, we will update the current upper-hull to include it. As we sweep across the x axis, there may be another new point that causes us to remove many points from our current upper-hull (Here, red dashed lines denotes the corrections that should have been made, while the black dashed lines represents the final convex hull):



So, we need to find a condition on when a new point is good and if not, then we need to update the found upper-hull. The central question is whether we turn left or right from the previous segment - if we're on the upper-hull, then turning right will always lead to the correct point. However, if we turn left, then this means that the penultimate point was not on the hull, and so we should go back to the previous point to define the hull. We need to keep dropping points until there are no left turns, in order to find the current upper-hull.

What is the runtime? For the actual algorithm, each point is added at most once, and deleted at most once, and so it just takes $O(n)$ for both the upper-hull and the lower-hull. Adding this onto the sorting time of $O(n \log n)$ time, the overall runtime is $O(n \log n)$. If the coordinates are integers, then counting sort results in a $O(n)$ runtime for convex-hull.

In fact, there is a deep sense that the convex hull problem is actually connected to sorting overall. It can be shown that a sorting problem can actually be converted to a convex hull problem, and a convex hull problem can be converted to a sorting one (as seen above), so these are actually equivalent problems.

There is also a case in which the convex hull may be simple and only contain a few points, and so it may be possible to get a faster runtime with an **output-sensitive algorithm**. Chan in '96 did so, finding a $O(n \log k)$ algorithm, where k is the number of points on the hull.

25.3.2 Halfspace Intersection Problem

In the **halfspace intersection problem**, we are given a set of halfspaces, and we're asked to draw the intersection. This is also equivalent to drawing the feasible region of a linear program.

To solve it, we can use a duality approach, where we map the point (a, b) as follows:

$$(a, b) \leftrightarrow L_{ab} = \{(x, y) \mid ax + by = -1\}.$$

With this mapping, we can show that points on a line map to lines through a point, and then the halfspace intersection problem simply becomes a convex hull problem.

This also has extensions in low-dimensional linear programming, since linear programming is just an extension of the halfspace intersection problem. If we find the convex hull, then we can find the intersection region, and then just run simplex on it directly (which is fast for low dimensions). It turns out that we can actually find linear-time (in the number of constraints) strongly-polynomial algorithms for linear programming in low dimension. Sweep lines can also be used to build a 'fascinating' data structure named **Voronoi Diagrams**, which we'll explore in the next lecture.

26 Lecture 26: Computational Geometry II

26.1 Voronoi Diagrams - Introduction

The Voronoi Diagram is a data structure useful for answering the question of finding a nearest point, specifically where we are given a set of points p_i in the plane and query points q_i of which we want to find the closest p_i. We'll define $V(p_i)$ to be the points in the plane that are closer to p_i than all other points. These subdivide the plane into specific disjoint regions, which we call **Voronoi regions**.

When we only have one point, then the Voronoi region of that point simply consists of the whole plane. When we have two points, the two Voronoi regions we get are half-planes, with a divider being the perpendicular bisector of thw two points themselves.

When we have three points, we still draw the perpendicular bisectors of each pair of points, but this creates 6 regions. This is a problem, since we're only supposed to have 3! In fact, this construction actually tells us the closest point as well as the second closest point. We can then combine the two regions at a time to create the Voronoi regions of the three points. In addition, these perpendicular bisectors will always meet at the same point, the circumcenter of the triangle, and we call this common point of intersection the **Voronoi point**.

When we add a fourth point, we can draw the perpendicular bisector and get even more Voronoi regions and points. What's interesting is that we now start to get closed Voronoi regions, i.e. polygons. These polygons themselves are intersections of halfspaces and are hence convex. In general, we will have n regions in the plane, which may or may not be open, as long as the diagram is **nondegenerate**, meaning that no four points are on the same circle (ensuring that no four lines intersect at a Voronoi point).

Now, given an instance of the nearest neighbor problem, we can actually solve it just by looking at the Voronoi diagram, and checking which Voronoi region it's in. To actually do the checking, we can use our persistent data structure all the way back from Lecture 3, which has a construction time of $O(n \log n)$, a space usage of $O(n)$, and a query time of $O(\log n)$, where this n is the number of segments on the Voronoi diagram.

So, we have to somehow relate the size of the Voronoi diagram to the number of points in it. We can make the Voronoi diagram planar by adding a **point at infinity** that will be connected to all the unbounded line segments.

Now, we can apply a famed theorem of Euler:

Theorem 26.1 (Euler's Polyhedral Formula)
In a planar graph (or polyhedron), then $V - E + F = 2$, where E is the number of edges, V is the number of vertices, and F is the number of faces.

Proof. We backwards induct. Consider a planar graph, and look for an edge whose endpoints both have degree ≥ 3. Remove that edge. Then, the number of edges and the number of faces both decrease by 1, and so the quantity $V - E + F$ is constant.

If we have a degree 2 vertex, then we can simply remove it and connect the vertices that it was originally connected to. The number of vertices and the number of edges both decrease by 1, and so once again the quantity $V - E + F$ is constant.

Finally, if we have a degree 1 vertex, then we just get rid of one edge and one vertex, which once again maintains the invariant.

If we apply these tricks, then we can reduce any graph into one with just two vertices with two parallel edges. By inspection, $V - E + F = 2$, and so any planar graph satisfies this. □

Now, let's apply the formula for the Voronoi diagram. We know by definition that $F = n$. We also know that each of the Voronoi points has degree exactly 3 (except for the point at infinity), and each edge has two Voronoi endpoints. Now, $2E$ counts each end of each edge, and this quantity is also counted by the sum of all degrees of all vertices, and so we have $2E \geq 3(V + 1)$.

Then, applying Euler's formula and substituting E, we see that $2(V + n - 2) \geq 3(V + 1)$, which upon solving, we find that $V \leq 2n - 7$, meaning that the size of the Voronoi diagram is linear in the number of points we have. Thus, that n from the persistent data structure solution is indeed equivalent to the number of points we have in the input.

26.2 Voronoi Diagrams - Construction

Once we can efficiently construct Voronoi diagrams, then we've finally solved the nearest neighbor problem. Like sorting, Voronoi diagram construction is one of the most studied problems, and it has about 17 different algorithms.

Note 26.2
The Voronoi Diagram is actually the dual of the projection of a lower convex hull onto a paraboloid. Essentially, each point gets projected upward to a paraboloid, and then find the convex hull there. Then, the lines that define the hull, when projected down, become the Voronoi diagram. This means that solving 3D convex hull can be used to construct the diagram.

We'll instead see another approach of construction, based on sweep lines. To stick with convention, we'll have the line sweep from top to bottom, and we'll build the Voronoi diagram at the sweep line, meaning that the Voronoi diagram above the sweep line should be accurate. However, we have the problem that a point below the sweep line can actually change the Voronoi diagram above the sweep line, and we need to take care of this.

Instead, we'll focus on the parts of the diagram that are actually guaranteed to be correct. Clearly, anything point closer to an input point (known as **sites**) over the sweep line are correct, since any additional point can only be further away. Thus, any safe region is defined by the set of points that are equidistant from a point and a line, which is just a parabola. If we compute the parabolas for every point with the sweep line, and take their union, then we end up with a boundary known as a **beach line**. Then, when we're constructing the diagram, anything above the beach line is guarenteed to be correct.

This algorithm is known as **Fortune's Algorithm**, and keeping track of the beach line is its main focus. When we only have one point, let us say (x_f, y_f), and the sweep line is at $y = t$, then the parabola we have is defined by the equation $(x - x_f)^2 + (y - y_f)^2 = (y - t)^2$, by definition. At a particular fixed x, if we differentiate, we obtain the equation

$$2(y - y_f)\frac{dy}{dt} = 2(y - t)\left(\frac{dy}{dt} - 1\right) \iff \frac{dy}{dt} = \frac{y - t}{y_f - t}$$

This means that the parabola with the lowest focus descends the fastest. When we reach the second point, we start out with a degenerate vertical parabola, which then widens out to a normal parabola as t decreases and joins the beachline. Now, let's consider the intersection of these two parabolas. By definition, the distances from those points to the foci are equal to their distances to the sweep line, for both parabolas. Since the distances to the foci are the same, this means that their intersection is going to travel on a straight line, which is their perpendicular bisector! Then, the beachline will be defined by the two parabolas, with the second (lower) parabola used between the intersection points.

When the sweep line hits a site, we call these a **site event**. But there are also other interesting site events. At a certain point, specifically, a Voronoi point, the perpendicular bisector is going to stop. This Voronoi point intersection happens on the beachline exactly when a third parabola appears and passes through that same intersection, and so this method of Voronoi diagram construction follows directly from geometry! As the sweep line goes further down, the beach line is once again defined just by two parabolas. Since the higher the focus, the slower the parabola descends, the parabola with the highest focus will not be part of the beach line after a Voronoi point is made. This is known as a **circle event**, since the Voronoi point is on the circumcircle of the foci.

Thus, we have site events and circle events, which respectively add a parabola and remove a parabola (and also generate a Voronoi point). In order to implement this, we need to track these events, and we can do so with a priority queue on each time. Each site event happens when the sweep line passes $t = y_f$. Further, we can predict the time of the circle event, since this happens at the bottom of the circle that goes through all three foci. We can also keep track of the location of each change, by keeping a BST on the parabola pieces of the beach line.

Our implementation for site events is then to search in the parabola BST, split the parabola there, and then add our new one. This may create potential circle events, and so we add these into the priority queue. We can also remove the previously adjacent trios of parabolas that are no longer relevant from the priority queue. When a circle event happens, we remove the parabola, and likewise update the circle event priority queue. Since each parabola is only involved in a constant number of priority queue updates at each event, then the work done for construction is only $O(n \log n)$. Thus, we can solve the nearest neighbor problem in $O(n \log n)$ time, $O(n)$ space, and $O(\log n)$ query time.

If we draw an edge between two points sharing a Voronoi boundary, then we end up getting a dual of the Voronoi diagram. Since each vertex has degree 3, then all faces in the dual are triangular, and so we get a triangulation of the planes, which are very important in PDE solving and computer graphics. The dual of the Voronoi diagram is the 'best triangulation,' which is known as the **Delaunay triangulation**. This 'bestness' comes from the fact that there are no long skinny triangles, which would be bad for approximation. And we get this for free with a Voronoi diagram construction!

27 Lecture 27: Online Algorithms I

27.1 Introduction to Online Algorithms

Previously, we've talked about algorithms that simply receive an input and compute the output from the input. Now, we'll focus on **online algorithms**, in where we get the input a little at a time, and where we need an immediate response, before we see the rest of the input.

Example 27.1
The stock market is a good example of needing an online algorithm, since we obviously don't know how it will change in the future. Another example is **paging**, in where we need more memory than is actually there, and the OS has to decide which pieces of memory to move to disk and which to keep in the main memory. When we have to read the memory from disk, we get a **page fault**, i.e. a slow fetch, and this causes program slowdowns.

To model these problems, we will define an input sequence $\sigma = \sigma_1, \sigma_2, \dots,$ in where after each input, we have to produce an output or perform an action, and we want to optimize our cost of our outputs/actions, which we'll denote $c_{min}(\sigma)$. It's usually easy to compute $c_{min}(\sigma)$ if we're given the sequence σ in advance, but usually this is not possible.

It doesn't make sense to just look at worst case behavior, since sometimes the cost is always bad. Instead, we'll compare it to the best possible outcome given an input sequence.

Definition 27.2
An algorithm is **k-competitive on σ** if $c_A(\sigma) \leq k c_{min}(\sigma)$. An algorithm is **k-competitive overall** if this is true for all σ. An algorithm is **asymptotically k-competitive** if it is k-competitive up to an additive constant. We call k the **competitive ratio**.

27.2 Ski-Rental

The **ski-rental problem** is a problem where we want to ski for some number of days, and we can either rent or buy skis. It takes 1 unit of cost to rent skis, and $T > 1$ cost to buy skis. The idea is that if we know that we'll ski a lot, we should buy, but if we'll only ski a little, then we should rent.

Let's consider limiting cases first. Suppose we want to ski n times. If we always rent, then it costs us n dollars. If we just always buy, it costs us T dollars. The first algorithm is not competitive, while the second is T-competitive.

One possible strategy is to rent for d days, then buy on day $d + 1$. An algorithm that knows the future will either always rent, or always buys on day 1. In the first d days, if the futuristic algorithm is renting, then the competitive ratio is 1. On the other hand, if the algorithm buys, then our ratio keeps getting worse. Now, once we buy, this ratio doesn't get any worse, in either case. Thus, the worst ratio of our algorithm stops after the moment we buy. Our worst input is one that occurs when we stop skiing after day $d+1$. Then, the cost of our algorithm is $T+d$, and the cost of our adversary is $\min(d+1, T)$.

What is the worst ratio we get? If $d+1 \leq T$, then the competitive ratio we get is $\frac{d+T}{d+1}$, which is decreasing as d increases. On the other hand, if $d+1 \geq T$, then our competitive ratio is $\frac{d+T}{T}$, which increases as d increases. Thus, we should choose $d+1 = T$ to minimize, and the competitive ratio we get is $\frac{2T-1}{T} = 2 - \frac{1}{T} \leq 2$. Thus, this is a 2-competitive algorithm.

27.3 Selling Stuff

Suppose that we are given a good which we can sell, whose price varies on each day. If we don't have any constraints, then there is no competitive ratio, since the price could just tank/rise on the day after, which we can't predict.

However, if we know the max/min price of the good M and m, then we can always get a competitive ratio $\Phi = \frac{M}{m}$ by just selling at the price m. A better strategy would be to set a reverse price $\sqrt{Mm}$. We then sell the first time we go larger than $\sqrt{Mm}$, and if it never does, we sell at price m. Now, let's analyze. If there is an offer of $\sqrt{Mm}$, then our algorithm gets us a competitive ratio of $\frac{M}{\sqrt{Mm}} = \sqrt{\Phi}$. Otherwise, if there are no offers of this value, then the best possible offer had value $\sqrt{Mm}$. The competitive ratio is then $\leq \frac{\sqrt{Mm}}{m} = \sqrt{\Phi}$, and so this gives a $\sqrt{\Phi}$-competitive ratio.

Now let's consider the same problem, but where we can sell our good in pieces. One method that works is just to set two reserve prices, at $m^{2/3}M^{1/3}$, and at $m^{1/3}M^{2/3}$, and sell half of our good at each. Thus, half of our goods are sold within $\Phi^{1/3}$ of the optimal price, and so our competitive ratio is at least $\frac{1}{2}\sqrt[3]{\Phi}$.

Generalizing, we can put reserves at prices $m, 2m, 4m, \ldots, M$, and so there are $\log \Phi$ reserves. We then sell $\frac{1}{\log \Phi}$ fraction at each of the reserve prices, and one of them is within a factor of 2 of the actual optimum. This means that we have a $2\log\Phi$-competitive ratio if we are allowed to divide our good into pieces.

But we can also use this method in our original problem, where we couldn't split up our goods! To do so, we simply randomize, and choose a price from $m, 2m, 4m, \ldots, M$ with probability $\frac{1}{\log \Phi}$. Then, the math is the same as above, and this will give us an expected $2\log\Phi$ competitive ratio.

Remark 27.3. *A variant of this problem occurred on the 18.410 midterm of Spring 2020.*

27.4 Online Scheduling

Let's consider $P||C_{max}$ again. We saw earlier that greedy scheduling gave us a 2-competitive algorithm, and this is online as well, so this also gives us a 2-competitive online scheduling algorithm. But remember, we did much better previously - we got a $\frac{4}{3}$ approximation if we scheduled in decreasing order, and got a PAS with an enumeration algorithm. We obviously can't do these techniques, since we don't know which order they arrive in!

But we can still do better than 2, which was shown in 1995 by Bartal, about 35 years after greedy scheduling was published. The idea is to plan for the arrival of a large job, since this broke our greedy algorithm. Bartal did so formally by splitting the machines into two sets - underutilized and overutilized machines. As large jobs arrive, the threshhold for underutilization/overutilization changes, and so he guaranteed that there always is an underutilized machine avaliable for scheduling. This lead to a $2 - \frac{1}{70} \approx 1.986$ competitive ratio.

This was rapidly improved on, in 1996, where we split the machines into multiple groups. Prof. Karger worked on this and was able to make an LP to optimize group sizes, leading to a 1.945-approximation. In 1997, this was improved to 1.923 by Albers, with an even messier algorithm. On the other side, a lower bound has been proven - there are no $\frac{4}{3}$-competitive algorithms.

27.5 Paging

Paging generalizes beyond computer memory - in general, we have a small fast memory and a large slow memory, and we need to decide what is where, moving things around as needed. The idea is that the memory is divided into pages themselves, and when we need something on a page, we fetch the whole page, which relies on **locality of reference**. When a page fault occurs, we need to decide which page we want to evict.

One heuristic is to evict the least frequently used page. Another heuristic is the least recently used heuristic. Other possible heuristics are first-in first-out (FIFO) and last-in first-out (LIFO). One that isn't possible but is optimal is to evict the one that's going to be used the farthest in the future, known as **Belady's algorithm**. It turns out that the least frequently used and LIFO heuristics are not competitive. On the other hand, least-recently used and FIFO heuristics are k-competitive, where k is the number of pages that fit in the memory.

We can show that least-recently used is $k+1$ competitive easily, and proving this for FIFO is similar and left as an exercise. We break the sequence of page requests into phases of $k+1$ distinct requests, and since we only have k space, each leads to at least 1 fault in the optimal algorithm. This trivially shows the claim. We'll prove the tight bound of k next time.

28 Lecture 28: Online Algorithms II

28.1 Paging Continued

Let's now show that the least-recently used heuristic is k-competitive. We likewise divide into phases, but of $k+1$ requests in the first phase, and k in the rest. At the start of phase 2, we know that the $k+1$th request is in memory. Now, let's analyze the second phase, which consists of requests $k+2$ to $2k+1$. If there is no fault until the $2k$th request, then we must have needed all the elements already in memory. This means that the $2k+1$th request causes a fault. Thus, there is a fault in the all-but-last request of the sequence, or one in the last request itself. This means that the optimal algorithm is guaranteed to fault at least once per phase, and this leads to a k-competitive algorithm with LRU.

Theorem 28.1

There are no α-competitive paging algorithms with $\alpha < k$. In other words, LRU is optimal.

Proof. Suppose that we have a set of $k+1$ pages. Run the online algorithm, and at each step, request the missing page. This means that we miss every time.

What should the optimal offline algorithm do? Since it knows the whole sequence, on a miss, it should just evict the one that is used furthest in the future. This means that the next $k-1$ requests will be hits, and so we only miss once every k requests in the optimal, meaning that we can make any online input be at least k-competitive. This shows the claim. $\square$

Note 28.2

Surprisingly, these algorithms were made by Sleator and Tarjan, the same people who designed splay trees. In fact, we can even interpret data structures as online algorithms - we have sequences of requests in data structures. Further, we have easy and hard input sequences for data structures, and we want to do as well as possible for any arbitrary input.

Now, caches are used all throughout modern computers, and having a k-competitive algorithm when dealing with lots of memory seems kind of bad. So, we've come up with different ways to characterize our algorithms. One is the probabilistic model, which simply states that each page has some certain probability of being asked for. Another way to is to consider resource augmentation, in which we make up for not knowing the future by increasing avaliable resources. In fact, it can be shown that we use k pages and the optimum has h pages, that we can get a $\frac{k}{k-h+1}$-competitive algorithm. In the case that $k = 2h$, we get a 2-competitive algorithm, if we work with double the memory, which sounds a lot better than the k-competitive one we had before.

28.2 Adversarial Inputs

How did we generate the lower bound? We used an adversary, which designed a sequence based on simulating the actual algorithm. Essentially, the adversary knew what we were going to do, and so it made an input designed specifically for exhibiting bad behavior here. To get around this predictability, we randomize. In many algorithms, adding some randomization significantly increases the expected competitive ratio that we obtain. We'll define a **k-competitive randomized algorithm** as one where the expected cost for *any* sequence is less than k times the optimal, plus a constant. Note that this is averaged over the random choices we make, not the input sequence!

One method for implementation is when we 'flip coins' when running the algorithm. Another is where we have a probability distribution over deterministic algorithms, where we use probability to decide which algorithm to use. If we decide the deterministic algorithm before the input, then it is actually not random.

We can classify adversaries based on what they know. An **Oblivious Adversary** is the weakest one that knows what our probability distribution is, but not our outcomes. A **Fully Adaptative Adversary** is one that knows the exact results of your randomization, and this is effectively deterministic. An **Adaptive Adversary** is one that knows your randomization up to the present, but not of the future.

28.3 Randomized Paging

For paging, it can be shown that against an adaptive adversary, that the best bound we can get is still k-competitive. However, for an oblivious adversary, we can do much better. One very, very, simple algorithm for paging is to simply evict a random page. It turns out that this is k-competitive, but has the advantage over LRU that we use no memory. We can do a lot better if we combine LRU and random eviction, which Fiat did in the 70s, known as the **marking algorithm**. Initially, all the pages are marked. When there is a fault, we first check if all our pages are marked, and if so, then we unmark all of them. Then, we evict a random unmarked page, and fetch the requested page. Then, no matter whether we had a fault or not, we mark the new requested page. In essence, this is an approximation to LRU - marking essentially tracks which pages have been requested recently. But with this bit of randomization, we get a significantly better ratio:

Theorem 28.3
The marking algorithm is $O(\log k)$-competitive.

Proof. We will divide into phases, as usual. Each phase will start on the first request and end on the $k + 1$th request, meaning that there are k distinct requests per phase. At the start of the phase, all the pages are unmarked, by definition. Note that the phases are defined by the input, rather than by random choices. Further, when a page is marked, it will stay in memory until the phase ends. This tells us that we miss at most once per page, per phase. Thus, we can ignore all requests after the first for each page in the phase.

We will say that phase i starts with having pages S_i in memory. We will define a request to be **clean** in phase i, if it's not in S_i. The intuition is that these clean requests generally are for pages that had came long before, and so the optimal algorithm would probably also fault here. On the other hand, we will define a request to be **stale** if it was in S_i. Then, we fault only if we evicted the page from S_i before the request came in, which is not likely since we evicted random pages.

Now let's do a formal analysis. Suppose that in the phase, we have s stale and c clean requests. All of them became marked, and all of them are still in memory. Then, consider the next request - if it's clean, we fault, and if it's stale, then the probability that we fault is equal to the probability we evicted it before the request came in. At this moment, all the stale requests must be in memory, either if it was there originally or if it was brought back after its first eviction. The rest of S_i are the candidates for what can be missing in the memory. Thus, c of $k-s$ candidates from S_i are missing, and so each request is missing with probability $\frac{c}{k-s}$.

For the whole phase, let's say we have c_i clean requests, meaning that we have $k - c_i$ stale requests. We pay a cost of c_i for the clean requests. For our analysis, we can assume all the clean requests happen first, since this is the worst case scenario. Then, the first stale request misses with probability $\frac{c_i}{k}$, the second with probability $\frac{c_i}{k-1}$, until the last one, which we miss with probability $\frac{c_i}{k-(k-c_i-1)}$. The sum of all of these probabilities, the expectation of the number of misses from the stale requests, is $c_i(H_k - H_{c_i})$, where H_n denotes $\sum_{j=1}^{n} \frac{1}{j} = O(\log n)$.

For our comparison to the optimal, we'll use a potential function Φ_i, which we'll define as being equal to the number of differences between the caches of our algorithm and the optimum, at the start of phase i. It's easy to see that this satisfies the definition of a potential. Then, when we get c_i clean requests, we know that there are at least $c_i - \Phi_i$ not in OPT's cache as well. At the end of the round, our algorithm has the k requests of the round. On the other hand, the optimum is missing Φ_{i+1} of those requests, meaning that they were evicted during this phase. This tells us that the optimum must have had at least Φ_{i+1} faults during this phase.

This means that the optimum had at least $\max(c_i - \Phi_i, \Phi_{i+1}) \geq \frac{c_i - \Phi_i + \Phi_{i+1}}{2}$ faults. Over all phases, we then have at least $\frac{1}{2}\sum c_i$ faults, which leads to an $O(\log k)$ competitive ratio. □

It turns out that this competitive ratio is actually optimal. We'll prove this next time.

29 Lecture 29: Online Algorithms III

29.1 A general technique for lower bounds

We usually don't talk about much about lower bounds, because they're 'depressing.' But we'll look at this one very general technique for doing so, and will allow us to show that the randomized paging algorithm from before is indeed a lower bound. We'll treat the algorithm as a game between the algorithm designer and an adversary, and we'll define the score to be the expected cost for any input sequence, minus k times the optimum for that sequence. Note that if this quantity is ≤ 0 for all inputs, that we've achieved a k-competitive algorithm.

This can now be interpreted as a zero-sum two-player game between algorithmist and adversary, with the payoff being equal to the score defined above. With deterministic strategies, if your opponent knows your strategy, then they'll change their strategy to beat yours, and this can go on forever (i.e. no equilibrium is reached). Instead, we'll use randomized strategies - then, we can disclose our strategy and still be able to achieve a good score.

Example 29.1
Let's briefly examine the game of rock-paper-scissors. There are no determinstic strategies. On the other hand, a randomized strategy for both players of choosing each option with probability $\frac{1}{3}$ is an equilibrium strategy, in that neither player changing will improve their score. This is known as a **Nash Equilibrium**.

To formalize, we will say player 1 has strategies i and player 2 has strategies j, which are deterministic strategies, also known as **pure strategies**. We will also define the **payoff matrix** M_{ij}, which has elements a_{ij} that tell us the payoff if player 1 plays i and player 2 plays j. If we then define the i-element vector x and j-element vector y, representing the probability of each player picking strategy i or j, strategy, the expected payoff is simply $x^{\intercal}My$.

Now, if player 1 knows y, then they will try to choose the best response x. In other words, they want to choose a strategy x such that the score is minimized - $\min_x \max_y x^{\intercal}My$. Similarly, if player 2 knows y, then they want to pick a strategy to maximize the score $\max_x \min_y x^{\intercal}My$. Note that the player who chooses second can achieve the optimum with a deterministic strategy.

Theorem 29.2 (Minimax Theorem)
In a 2-player zero-sum game, these quantities are equal: $\max_x \min_y x^{\intercal}My = \min_x \max_y x^{\intercal}My$.

Proof. By strong LP duality. See old PSET. □

Now let's apply this to online algorithms and adversaries. This is a two-player game, where one randomly chooses the strategy A from a probability distribution and other chooses the input sequence σ. Then, the score is equal to $\min_{randA} \max_{\sigma} \mathbb{E}[C_A(\sigma) - kC_{OPT}(\sigma)]$. Since deterministic strategies perform just as well as randomized ones when the minimax theorem, and by the minimax theorem, this quantity is also equal to $\max_{rand\sigma} \min_A \mathbb{E}[C_A(\sigma) - kC_{OPT}(\sigma)]$. Translating this into English, we get the following:

Theorem 29.3 (Yao's minimax theorem)
The best competitive ratio achievable by a randomized online algorithm against an oblivious adversary, is equal to the best competitive ratio achievable by a deterministic online algorithm against the worst known distribution over inputs σ.

This now gives us a tool to prove lower bounds!

Corollary 29.4
If there is a distribution over σ such that no deterministic algorithm can do well, then we can conclude that there is no randomized algorithm that can do better.

Remark 29.5. *Infinite input sequences may cause some problems, but mathematicians have figured it out and it works out in the end. As computer scientists, we don't think about infinity, and just hope that everything works instead. On the bright side, if there is a bad input, it's usually finite.*

29.2 A lower bound on Paging

One possible bad input to paging is just to choose randomly over all possible inputs of some length. We'll take the length of the sequence to be $k+1$ pages, since k pages perform extremely well. Regardless of what's in memory, the probability of a fault is equal to the probability that a wrong page is requested, which is $\frac{1}{k+1}$. Thus, every deterministic algorithm has at least an expected number of page faults equal to $\frac{1}{k+1}$ per request, meaning that we need $O(k)$ requests in expectation to see a fault.

OPT instead gets to see the whole sequence σ in advance, and will evict the page that is the farthest in the future, and won't fault until the next request for that page arrives. Since the sequence of requests is random, we will see all $k+1$ items before we get a fault. This is the **coupon collector problem**, and has the well known solution of needing expected $O(k \log k)$ requests. Thus, the deterministic algorithm has $\log k$ times as many faults as the optimum, which shows the lower bound for randomized paging!

29.3 The k-server problem

The k-server problem was introduced by Manasse in 1988, and it involves k servers which move between points on a metric space. Each request is a point in space, and we must move some server to that point. Over a large sequence of requests, the total cost of serving the requests is equal to the distance travelled by these servers. It's a relatively general problem, and can, for example, be used to model paging on a uniform metric. A server would then be on a page, if the page itself is in memory, and when a server is moved, this is equivalent to evicting the page. It can also be used to represent other problems, such as weighted paging and optimizing selecting heads on hard drives.

It was a hard and poetic problem and so theoreticians spent a lot of time on it. The first algorithm to try is a greedy algorithm; unfortunately, it doesn't work. If we have two points A, B close to each other and a point C far away, with the servers starting on A and C, and we have the input sequence $\{AB\}^n$, then we get an infinite competitive ratio. But we can still get some insight from the failure of the greedy algorithm. This is very similar to a ski-rental problem, in that we can have many small costs or just one large cost.

Ultimately, we can't just move the servers, since we have many servers and don't know which to move. A better one would be to probabilistically weight it by distance - each server moves with probability inversely proportional to the distance. This is known as the **Harmonic Algorithm**, and is $O(k^k)$-competitive. This was a big improvement over an $O(k \uparrow\uparrow k)$ algorithm presented by Manasse in his paper introducing the problem.

Another possible algorithm defines a work function, in where we essentially track the offline optimum, and try to converge back to the current stage of the offline optimum. If we define $W_i(x)$ to be the optimal cost of serving the first i requests and finishing with the servers in the state x, then we want to minimize over all states x of $W_i(x) + d(X_{i-1}, X_i)$. The algorithm has an abysmal runtime due to the computation of the offline optimum, but this was shown to be $2k$-competitive in 2001, a huge improvement!

Let's now examine a special case of this problem, where all the servers are on a line (an 'on line algorithm'). If we get a request between two servers, then it makes sense to only move servers that are closest to the request point in either direction, and this can be shown by a simple swapping algorithm. We don't know which one is optimal, so we'll move both servers the same amount towards the request, until one reaches it (the **double coverage algorithm**). If the request is on one side (outside the convex hull), then we only move one server, since it would make no sense to move two. Locally, we spend at most twice the optimum cost. In fact, this can be proven to be k-competitive, which we'll do next time.

30 Lecture 30: Online Algs. IV, External Memory Algs. I

30.1 Double Coverage Algorithm

Theorem 30.1

The double coverage algorithm for on-line k-server is k-competitive.

Proof. As usual, we'll use a potential function. We want this to measure our inoptimality, somehow capturing how different DC is from the OPT. One metric is just the total cost to move the DC servers to the positions of the optimum, and we can find this with a min-cost matching algorithm, which we'll call M. In our analysis, we will also need a second parameter d_{ij}, which is simply the distance between the two servers i and j. Then, our potential function will be $\Phi = kM + \sum_{i<j} d_{ij}$.

For our analysis, we will first move the servers of OPT, and update the potential. Then, we will move the double coverage servers, and also update Φ. We want to show that if OPT moves a distance d, then Φ also increases by $\leq kd$. We also want to show that if DC moves a distance d, then Φ decreases by $\geq d$. This will show that our algorithm is k-competitive, since the potential is always nonnegative, and thus DC cannot decrease by $\geq kd$.

For the first claim, this is simple, since the sum term doesn't change. The only change is in M, which can increase by at most d, showing the claim.

For the second claim, we'll split into where the request is - either inside or outside the convex hull. If it is outside the convex hull, then we move the closest server. The potential increases by $(k-1)d$ from the distance term. For the matching, we need to do an analysis based on where each pairing is. If the server that moves in the optimum is matched to the one we move, then the potential decreases by kd, showing the claim. In fact, we should always swap the matching such that this is true, since this leads to a lower min-cost matching.

If the request is in the convex hull, then, the matching cost never increases. The one that is matched to the optimum decreases by d, while the other could increase by up to d. For the movement, since the two servers move in opposite directions, most of the distances actually cancel. The only thing that does change distance are the servers that we actually move, which decreases the potential by $2d$. Thus, Φ decreases by at least $2d$ in this case, which shows the result. □

This method generalizes to graphs on trees. This is somewhat important, since it has been proven that any metric is the sum of $O(\log^2 n)$ tree metrics. In fact, this leads to a randomized k-server algorithm that's $O(\log^3 n \log^2 k)$-competitive, where n is the number of points in the metric space, if we use randomization and tree metrics to guide our choices.

30.2 Introduction to External Memory Algorithms

The general idea is that RAM is fast, while disk access is slow. Some algorithms, in fact, have this as a limiting factor - the algorithm runtime is proportional to the number of disk accesses, and hence we want to minimize the number of disk accesses for a fast algorithm. Like we did in caching, we will pull a block of information at a time from memory in order to exploit locality.

For such problems, we will deal with a block size B, a problem size N, and a memory size M, the number of items we can fit in the fast memory. One basic algorithm is for scanning arrays on disk. This takes $O(N/B)$ time for standard operations, since we pull in B items at a time and read each once. In general, any array algorithm will take this time.

Now let's increase our dimension, and deal with two $N \times N$ matricies (so our problem size is N^2). We need to first find out how we store our matricies in the blocks, and one way to do so is in **row-major order**, where we essentially put some number B of elements in a block per row, for each matrix. Overall then, our runtime is $O(N^2/B)$ to access. Addition has the same runtime, since we simply add the respective elements of the matricies.

Multiplication is much harder. If we use the naive algorithm for multiplication, then we need to read a row, read a column, and output one value. Reading a row takes time $O(N/B)$ while reading a column is $O(N)$ (since each column is in a different block). Thus, each multiplication takes $O(N)$ time, and this thus takes $O(N^3)$ time (block reads).

Instead, if we stored our second matrix in **column-major order**, then accessing the second matrix is also $O(N/B)$ time. This leads to overall $O(N^3/B)$ time. However, there is a predictability problem, in that we don't necessarily know beforehand which matrix is on the left and right.

There is a further problem, in that we are 'wasting' our row and column reads. We're inefficiently reading each row and column N times, which is N times too many. We're going to need a different method, which we'll explore next time.

31 Lecture 31: External Memory Algorithms II

31.1 Matrix Multiplication Continued

We saw that we could get a $O(N^3/B)$ algorithm simply by storing the matricies in row-major or column-major order. In fact, we can do better, by using a block decomposition of matricies - using one square submatrix as a memory block. The submatrix should have size $\frac{1}{2}\sqrt{M} \times \sqrt{M}$ in order to fit in to memory. Then, we need to read $O\left(\frac{M}{B}\right)$ blocks to multiply one pair of these block matricies. To get the actual output block matrix, we need to sum over $\frac{N}{\sqrt{M}}$ block matrix multiplications, and we have $\frac{N^2}{M}$ total matrix outputs. This means that our runtime becomes $O\left(\frac{N^3}{M^{3/2}} \cdot \frac{M}{B}\right) = O\left(\frac{N^3}{B\sqrt{M}}\right)$ time. With a large memory then, we can get significant improvements in runtime.

Remark 31.1. *These algorithms were based on the $O(N^3)$ sequential algorithms of matrix multiplications, but new faster ones have been developed - of runtimes approaching $O(N^{2.376})$. These can be externalized the same way.*

31.2 Linked Lists

Let's now think about how we can use external memory techniques to simulate a linked list, specifically supporting an insertion or deletion operation, when we are given the current position. We can do this by reading the block, deleting the element, and then updating the previous element and the next element. These may be in different blocks, so we get up to 3 reads and 2 writes per operation, which seems excessive, but is $O(1)$ anyways.

For our scan operation, we will need $O(1)$ per item. However, if we store our elements contiguously in memory, then we can scan B items per block, and our amortized scan will cost only $O(1/B)$ per item. However, the problem is that insertions and deletions can change the structure of the memory operations, and after many operations, the internal structure is lost.

To get around this, we need to augment our previous operations to maintain the structure. For deletion, one idea is just to leave a 'hole' in the block, which would maintain the block structure. But this doesn't work out, since we can delete $B-1$ elements from each block, and each block would only have one element, which is bad. A better idea is to leave holes until the block is less than half full, then combine it with the next block. There are two possibilities in this case - when the second block is less than half filled, then we can just combine them. Otherwise, if it has greater size, then we can just balance the blocks such that both are more than half filled. This works and means that each block still has $\Omega(B)$ items and we only have to do $O(1)$ block reads for deletion!

For insertion, if there is a hole created by our deletion, then we can simply insert into the hole. Otherwise, we can split the block into two blocks of size more than half full, artificially creating a hole which we insert into. This maintains the invariant that our blocks still contain $\Omega(B)$ elements and so both operations can be implemented in $O(1)$ time. Scanning then reduces to time $O(1/B)$ per item.

31.3 Search Trees

One way we can implement search trees is simply to move the whole thing into external memory, getting us $O(\log N)$ reads and writing. But we can do better, by taking advantage of the block size. If we structure all our blocks carefully, breaking into size-B trees with height $\log B$, then to get to the bottom we only need $\frac{\log N}{\log B}$ reads, which is also our runtime.

A more canonical way to think about this is through trees with B children at each layer, which we conveniently denote as **B-trees**. These are "the most important data structure in external memory algorithms." Their height is $\log_B N$ which is the same runtime that we got before.

To implement them, we will maintain the invariants that all the leaves are at the same depth, that all keys are at the leaves, and that the nodes are relatively full (i.e. more than $B/2$ items, except for the root node). The internal nodes will then not contain values to return (but only where to search), but rather just serve as splitters.

How do we maintain insertion and deletion while maintaining these invariants? For insertion, if the leaf block isn't full, then we can simply insert there. If the leaf block itself is full, then we simply split into two leaf blocks of equal size, and then copy the middle key up to the parent as the spltiter. If we reach the root, then we can also split the root itself - this way, the tree gets deeper with an insertion. The work is $O(\log_B N)$ since we simply need to do tree traversals.

The main problem with deletion is the same one we had with linked lists - that some blocks may become too empty. And we can do the same solution. If a block is half empty, then we merge it with a neighboring block, and if it eliminates a block, then we need to delete a splitter from the parent. Recursing on the tree structure, propogating up, this takes $O(\log_B N)$ work, and may cause the depth to decrease if the root's children are merged.

Note 31.2

This idea also works with binary trees, and a special variant of B-trees that is sometimes used is known as a **2-3 tree**, where each node has degree 2 or 3.

31.4 Sorting

As usual, we can just import a standard algorithm, and see how it performs in external memory. For quicksort, each scan takes time $O(N/B)$ and we need about $\log(N/B)$ time overall, which has total runtime $O\left(\frac{N}{B}\log\frac{N}{B}\right)$. If we use merge-sort, we get the same runtime as quicksort. If we use our B-trees from above, we use $O(\log_B N)$ time for each element, for a total runtime of $O(N\log_B N)$.

We're seeing something glaringly missing in our runtimes - the memory parameter M. If we can take advantage of M then our runtimes should be able to be improved significantly. One easy improvement we can do is just to scan and sort everything in memory once we get to M items. Then, the runtime becomes $O\left(\frac{N}{B}\log\frac{N}{M}\right)$ for quicksort and mergesort. This is significant as $M \gg B$.

There turns out to be a better way to use the memory in mergesort. Instead of a 2-way merge, we instead do a M/B way merge. We take the M/B sorted lists, merge them into one list with one scan. We put the front of each list in one memory block, and then keep emitting the min item to the output block, which we then write out. When an input list is empty, then we feed the next one. The time it takes is simply equal to the number of data blocks, and thus the runtime is only $O\left(\frac{N}{B}\log_{M/B}(N/B)\right)$. This is actually information-theoretically optimal, and so we *can't do better*.

One point of interest is that in standard memory, we need $O(\log N)$ time to search and $O(N\log N)$ to sort. On the other hand, we need $O(\log_B(N))$ time to search and $O\left(\frac{N}{B}\log_{M/B}(N/B)\right)$ time to sort, and so there is somewhat of a mismatch between these two cases. Specifically, if we have this sorting algorithm, then we should be able to do a search operation in time $O\left(\frac{1}{B\log(M/B)}\right)$. This is nonsensical since we have to read a block at a time, but there is a data structure known as a **Buffer Tree** which actually does this. It gets around the impossibility, by batching up search requests and then having this cost as an amortized cost.

Finally, there is the idea of **cache-oblivious algorithms**. The algorithms that we've talked about before all assume that we know the block size B. But in real life, we need to optimize the algorithm such that all parts of the memory hierarchy are optimal. This is somewhat tedious to do manually, but the cache-oblivious algorithms introduced by Leiserson (another member of the department) don't need either B or M and can still get the same bounds! He was able to get the same bounds for all the problems we discussed above with such algorithms.

Example 31.3
Consider matrix multiplication. When we multiply two $N \times N$ matricies, we will split it into four $N/2 \times N/2$ matricies (per original matrix) and recurse, rather than $\sqrt{M} \times \sqrt{M}$ ones. The algorithm itself can tell when it stops reading from external memory, and the runtime turns out to be the same as ours above.

32 Lecture 32: Parallel Algorithms I

32.1 Definitions and Models

Parallel Algorithms, as a theoretical topic, started to flourish in the 80s, but then somewhat died in the 90s since the theoretical models didn't really match the practical results. However, now, they are seeing a resurgence and are interesting in their own right and can bring insight as well.

There are two different models of computation traditionally associated with the field - **circuits** of logic gates, and **parallel processors** over shared memory. In the circuits model, performance is measured by the size - the number of actual logic gates, and the other is measured by the depth, or the number of layers. In the parallel processors model, performance is measured by the speed - the number of parallel steps needed, and also in the number of processors. It is easy to draw an analogy between the two models between the size/number of processors and the depth/speed.

In the circuits model, we'll start with the basic logic gates of AND, OR, and NOT. Nowadays, we use clocked circuits, in where the output is avaliable one cycle after the input. The depth then determines the time until the answer is avaliable, and so we obviouslly want to minimize depth.

Another factor to consider is known as **fan-in**, or the number of inputs to one gate. We can consider **bounded fan-in**, in where each gate has a constant number of inputs, or **unbounded fan-in**, where we can have arbitrary inputs. We can transform an unbounded to a bounded one by imposing a binary tree structure of 2-element gates themselves, with depth $O(\log n)$.

We define **AC(k)** to be all functions that are computatable with a circuit of depth $O(\log^k n)$ using polynomially many gates with unbounded fan-in. Similarly, we define **NC(k)** to be all functions that are computable with a circuit of depth $O(\log^k n)$ using polynomially many gates with bounded fan-in.

Joke 32.1. *NC actually stands for Nick's class, and was named that way by Steve Cook. There is also a class called SC, which stands for Steve's class, and was named by Nick Pippinger.*

Note 32.2

One might wonder where the intuition for these complexity classes comes from. The idea is that we want to compose algorithms, such that one algorithm calls others. So, if we have an algorithm that takes $O(\log n)$ steps, which also calls a subroutine that also takes $O(\log n)$ steps, then we will end up with a $O(\log^2 n)$ algorithm. For sequential algorithms, we see the same idea, except that our starting point for algorithms is usually linear runtime - hence why we care about polynomials.

Theorem 32.3

We have a simple relation between the complexity classes: AC(k) $\subseteq$ NC(k+1) $\subseteq$ AC(k+1)

Proof. The second relation is obvious. For the first one, we can simply expand out all our logic gates as per above, increasing the number of steps by at most a factor of $\log n$. $\square$

Finally, we define the class **AC** (which is also equivalent to **NC**) to be the union over all k of all $AC(k)$ or $NC(k)$ i.e. any polylogarithmic algorithm.

32.2 Addition

The most straightforward way to approach addition is with a **ripple-carry adder**. Essentially, this is the same algorithm as taught in grade school - add two bits, and carry the next bit over to the next digit if necessary. Unfortunately, this leads to a $O(n)$ depth and $O(n)$ size algorithm since each bit addition needs to be processed before the next.

We can do better with **carry-lookahead** adders, which have size $O(n)$ but depth $O(\log n)$. The idea is that if we know all the carry bits beforehand, then we can simply process all the actual additions in parallel and know the result without this chain of additions. If we preplan for hypothetical values of carry bits, then we should be able to speed up significantly.

Well, given an input bit addition of a_i and b_i, we additionally have the carry bit c_{i-1} from the previous gate. What are the possible values for the output carry c_i? If at least two of the bits is equal to 1, then $c_i = 1$. This means that if $a_i = b_i = 0$, then the carry is $c_i = 0$, which we call a **kill** operation. If $a_i = b_i = 1$ then the carry is $c_i = 1$, which we call a **generate** operation. Finally, if $a_i \neq b_i$, then $c_i = c_{i-1}$, which we call a propagate operation.

Now, we can treat each gate by their identities. The carry bit, and hence the result, is then computable by chaining the result of many of these gates. There is a simple multiplication table for these operations:

	k	p	g
k	k	k	g
p	k	p	g
g	k	g	g

The zeroth gate is a kill gate, since the carry to the first gate is always 0. Then, the nth gate is simply the composition of all the previous gates. Note that this composition is never p! (proof: by induction) This means that simply by computing these compositions of gates, we can directly know all the carry bits. Each gate's i's carry will then be given by $x_i \otimes x_{i-1} \otimes \cdots \otimes k$, and they are collectively known (due to historic reasons) as **prefix sums**.

So, computing prefix sums efficiently gives us a way to add numbers quickly. One way we can do so is through a binary tree structure, adding the prefixes of two numbers at a time. If we have a binary tree for computing each prefix sum, then we get our desired $O(\log n)$ depth, even though we use $O(n^2)$ area.

A better way to go about this is to assemble any prefix from subtrees of one complete binary tree. For example, if we consider a binary tree on 8 elements at the leafs, and we want to find $x_6 \otimes x_5 \otimes \cdots x_0$, then we can simply compute this via $(x_6) \otimes (x_5 \otimes x_4) \otimes (x_3 \otimes x_2 \otimes x_1 \otimes x_0)$, where each parenthesized quantity is one that we've already computed along the tree.

In practice, in order to avoid excessive fan-out, we do so by starting from the right, feeding the subtree values up, and then down - passing up the product of all children, and passing down the product of the right subtree to the left child for use in the left subtree wherever needed. In either case, this reduces to $O(n)$ gates while keeping the $O(\log n)$ depth.

32.3 Parallel Processors/PRAM

As mentioned above, the PRAM model relies on a set of parallel processors along with a shared memory. Each processor operates on synchronous cycles, and this leads to problems with memory read/writing. For reads, it seems to be generally fine for many processors to read the same location. However, since sometimes (physically) the memory value cannot be delivered to each processor in the same cycle (due to excessive fan-out), we divide into exclusive and concurrent read models for theoretical analysis.

For writes, we have the same problem, and so we likewise divide into exclusive and concurrent write models. When we actually have a concurrent write, there are many ways of dealing with it - choosing an arbitrary result, choosing the max/min, or even just writing garbage. Anyways, combining these possibilities we have the CRCW, CREW, or EREW models of computation, with the CREW (concurrent-read exclusive-write) model being preferred since we don't need to deal with write conflicts.

To change these models to a complexity class, we define **CRCW(k)** to be problems solvable in $O(\log^k n)$ time using polynomially many processors under CRCW operations. We can define **EREW(k)** in a similar way, and like what we saw with circuits, we also have the relation that CRCW(k) $\subseteq$ EREW(k+1). This can be proven with the same method as we did with circuits, by considering a binary search tree.

Finally, we define the NC class to be equal to $\cup_k CRCW(k)$ over all k. The NC here is equivalent to the one we had with circuits - these definitions are indeed equivalent. Every depth d circuit and g gates can be simulated by a PRAM with g processors in time d, and conversely, any PRAM algorithm with n processors that run in time t can be built into a circuit with polynomially many gates and polylogarithmic depth.

Note that there is a difference between EREW and CRCW computational models. We can see this when we consider computing the OR of n possible values. In the EREW model, we need $\log n$ time by constructing a binary tree. In the CRCW model, we can set the initial output to 0, and have each processor check the value of the inputs. If any processor sees a 1, then they can just write a value of 1 to the output, and so this is $O(1)$ time, $O(\log n)$ faster.

33 Lecture 33: Parallel Algorithms II

33.1 MAX operation

The bound that we got before of $O(\log n)$ for computing an OR operation in the EREW model is actually a tight bound. To show this, consider a case where only a single value out of the n inputs is 1. Initially, each memory location only knows its own value, and each processor can combine the results from two locations in one step. We need a constant number of steps to know two values, then four, etc. which turns out to be logarithmic.

Now let's consider the problem of finding the maximum element. In the EREW model, we can do so in $O(\log n)$ with the same binary search tree model (and this is a tight bound as well, since we can use maximum computation to compute an OR as well). In the CRCW model, we get a problem compared to last time - we can't just have everything write to the same location, since overwrites will happen.

But we can still get $O(1)$ time if we use $O(n^2)$ processors. We do so by making each number check if it itself is the maximum, and if so, then write itself to the output. To actually check in $O(1)$ time, we need to compare a set number with every other number (which is $O(1)$ with n processors), and then take the OR of all the comparisons in $O(1)$ time as before. Only the number that has a value of 0 for the OR writes itself to the output.

Note 33.1

When we assign processors to tasks, we assume that each has an integer ID. Then, we can assign processors to different tasks by their unique ID, while they act synchronously.

Definition 33.2

We define the **work** of an algorithm to be the number of processors, times the time that it takes. Note that we can simulate a PRAM algorithm on fewer processors, and then work becomes a measure of time needed.

Further, we define an **efficient** algorithm to be one whose work is bounded by the runtime of the best sequential runtime. Note that parallel algorithms' work must always be as slow as a sequential algorithm, since otherwise we could get a faster sequential algorithm. This is known as the "cost of parallelism"

Our previous definition of MAX had $\Theta(n^2)$ work since we used $O(n^2)$ processors. But we can get better work by using $O(n)$ processors, where we can achieve a $O(\log\log n)$ algorithm:

Suppose that we have k candidate maxes remaining. Then, we can make $\frac{k^2}{n}$ groups of $\frac{n}{k}$ items. Then, in each group, we run our $O(1)$ max from before, which needs $\frac{n^2}{k^2}$ processors per group. This is a total of n processors, as desired. So, with n processors, we can reduce the problem of finding a max of k items to one of $\frac{k^2}{n}$ items in $O(1)$ time. If we use $2n$ processors rather than n, then we will reduce the number of items at each iteration, and this recurrence solves to $O(\log \log n)$, as desired.

33.2 Parallel Prefix

We know that we can get $O(\log n)$ time for computing prefixes using n processors. This leads to $O(n \log n)$ work, while the sequential algorithm needs $O(n)$ time. We know we can't improve the actual time it takes (since otherwise we can compute OR faster), but maybe we can improve the actual number of processors, to $O\left(\frac{n}{\log n}\right)$.

To do so, let's make blocks of $\log n$ contiguous x_i. We assign one processor per block, and then compute y_i, which are the products of all its items. We can then run parallel prefixes on y_i. Finally, we can have each processor fill in the prefixes starting from the prefixes of y_i. Each processor does $\log n$ time on its own block to fill things in, and then parallel prefixes on y_i takes $\log n$ as well. We only need $O\left(\frac{n}{\log n}\right)$ processors to compute the prefixes on y_i, while maintaining the $O(\log n)$ bound.

33.3 List Ranking

Suppose that we have a linked list of n items, and we want to find its position in the list, or the number of items following it. The list itself is stored in an array of list nodes, but the memory locations are not contiguous or in order. Solving this would ultimately lead to a nice method to unfold a linked list into an array.

If we give each memory element a value of 1, then we simply want the parallel suffixes in order to find its position. Unfortunately, we can't apply our technique from before as we don't have an array to work with. Instead, we will use a more general technique known as **pointer jumping**.

In this technique, we have a value $d(x)$ in each node x, and we additionally have a next pointer $n(x)$. In our example above, $d(x) = 1$ for all x. The pointer jumping operator is going to increment $d(x)$ by $d(n(x))$ and change $n(x)$ to $n(n(x))$.

When we perform this operation, we maintain an invariant that the suffix sum for each node is unchanged. Further, this also breaks the linked list into two halves of equal length. $O(\log n)$ steps then are sufficient to shorten all lists to length 1 while perserving all the suffix sums, and so with n processors we can solve the problem with $O(n \log n)$ work.

33.4 Binary Search and Sorting

Let's first go over binary search. If we have 1 processor, we can trivially get a $O(\log n)$ runtime sequentially, and with n processors, we can get $O(1)$ runtime by looking at everything in parallel. If we have k processors, then we can get $O(\log_k n)$ items by splitting into k groups, and then checking each group in parallel. We can narrow it down to one group in $O(1)$ time, essentially turning it into a subproblem of reduced size $O(n/k)$. This turns out to be optimal.

For sorting, let's first consider what we can do with many processors. One trivial algorithm is to find the maximum in $O(1)$ and recurse, which gives us a $O(n)$ sort with n^2 processors. But this is bad - our standard parallel algorithms should have runtimes in NC.

We can get a smarter algorithm with the same idea that we had with max - by comparing each element to all other elements. If we count the number of items that are smaller than some specified item, we can easily sort our array. The total time it takes is $O(\log n)$ to count the number of smaller items, with $O(1)$ time for everything else, and so our total runtime is $O(\log n)$, though with $O(n^2)$ processors.

Let's think now about how we can use a mergesort style algorithm. We have $O(\log n)$ phases of merging, and so if we can make merging fast, then we can get a fast parallel algorithm. If we have two sorted lists, then we can merge them and find the actual position of where one element goes with a simple binary search. If we give each item one processor, then we can complete each merge phase in time $O(\log n)$ with just n processors. Then, merge sort will ultimately take time $O(\log^2 n)$ with n processors, which is much closer to our sequential bound.