6.854 Lecture Notes

THANADOL CHOMPHOOCHAN

Fall 2020

This file contains all the raw notes I took when I was taking 6.854 Advanced Algorithms in Fall 2020, taught by Karger. I barely edited anything before compiling it as this file, so there may be factual or typographical errors. Please use this at your own risk. Please email tcpc@mit.edu if you wish to provide feedback. *(Last updated: December 6th, 2020)*

Contents

1 Fibonacci Heaps

Fibonacci Heaps are developed by Michael L. Fredman and Robert E. Tarjan in 1984 to solve Shortest Path (SP) and Minimum Spanning Tree (MST) problems.

Recall, to solve MST, we can use Prim's algorithm which keeps adding the "closest" neighbor to the tree. This is similar to Dijkstra's algorithm.

To optimize runtime for Prim's Algorithm, we use a **priority queue** which supports three operations:

- Insert items: used n times
- Find/delete minimum: used n times
- Decrease keys: used m times

Using a standard binary heap, we get total runtime of $O(nm \log n)$. If we can optimize decrease-key operations to $O(1)$ per operation, then we will get a better runtime of $O(nm \log n)$.

Notice that the runtime of a heap is completely determined by the height of the tree. So, let us instead use d-heap where each node has d children nodes. The height would be $O(\log_d n)$. This gives faster insert and decrease-key runtime. However, delete-min now takes $O(d \log_d n)$. Prim's algorithm will take a total of $O(m \log_d n + nd \log_d n)$ runtime.

The best theoretical runtime we can achieve is when both $m \log d$ and $nd \log_d n$ terms are set equal: $m = nd$. Then, we get a runtime of $O(m \log_{\frac{m}{n}} n)$. If $m = n^2$ then the runtime is simply $O(m)$.

§1.1 Inventing Fibonacci Heap

We try to be **lazy**, meaning every time we work, it should count as much as possible. Imagine there is an **adversary** who tries to force you to work hard. We can force the adversary to spend a lot of effort to make us work, and amortize our work vs. theirs.

Let's see how far we can go with laziness.

- With insertion, we might use a linked list and simply insert stuff to the head of our linked list in $O(1)$. This might sound like the best we can do, but in fact we can just say "okay" and do nothing!

- With delete-min, we need to scan for minimum in $O(n)$, which averages out to only $O(1)$ per inserted item. *The problem occurs when we have to do multiple delete-min operations.*

The rule of thumb is: whenever we work, we simplify. In this case, notice that each comparison reveals smaller/larger elements. So, we can just ignore larger elements when we seek the minimum. Represent this as trees where the loser is made the child of the winner, and we get a heap-order tree. In fact, we will get a "caterpillar tree."

Definition 1.1.1 (Caterpillar tree)**.** A caterpillar tree is a tree where there is one main chain the tree, and other nodes are connected to these main nodes only.

Delete-min breaks the tree into subtrees. Only roots of the subtrees are viable candidates for the next minimum. The problem: if first element is the minimum, then we get a star graph. (That one node plays often and wins every time.) The runtime for delete-min is the max number of children. If we can figure out a way to keep the maximum degree small, then we will have a fast heap data structure.

One way to do this is to run a competition where no nodes get to play too many times. Basically, run a bracket tournament: put nodes into pairs, compete, have winners compete, repeat until we have one. We will get a **binomial heap**. We call this a binomial heap because the number of the nodes in each level of the tree maps out the binomial coefficients. Its properties are:

- A height h binomial tree has a total of 2^h nodes.
- The maximum degree is h.
- For a heap with n items, the max degree is $\lg n$.

We must generalize this data structure to interleave insertions and deletions. What we can do is

- Record each node degree (like a tournament record).
- Only compare nodes which have the same degree (same tournament round). This is the **union by rank heuristic**.

Consolidation:

- Bucket the nodes by degree.
- From degree 0 and up, link pairs in the same bucket and promote to the next bucket, until there is at most one in tree in the bucket.

Doing this results in at most $O(\log n)$ trees being created.

We are going to combine with **lazy insert**, which simply adds a heap-order tree with degree 0 to the collection. Keep a pointer to current minimum value. On **delete-min**, remove the minimum, add children to the collection, then do the consolidation to find the next minimum.

The worst case runtime of delete-min is $O(n)$ which happens when do insert n times and do delete-min. However, this has amortized $O(1)$ time complexity and simplifies the data structure. In general, delete-min cost is the number of trees in the collection plus the children of the root.

§1.2 Potential function

Definition 1.2.1 (Potential function)**.** The potential function ϕ indicates the level of complexity in our data structure. Whenever the adversary works, ϕ increases. When we work, ϕ reduces. Let a_i be the amortized cost of operation i and r_i be the real cost of operation i. We have

$$a_i = r_i + \phi_i - \phi_{i-1}.$$

The total cost $\sum a_i = \sum r_i + \phi_{\text{final}} - \phi_0$. If $\phi_f \geq \phi_0$, then $\sum a_i \geq \sum r_i$. It is sufficient to set ϕ_0 and check that $\phi_f \geq 0$.

For heap, we can set $\phi(D_i)$ to be the number of heap-order trees. We have $r_i = \#(\text{HoTs}) + \#(\text{children})$. We want to calculate:

$$a_i = \#(\text{HoTs}) + \#(\text{children}) + \phi_f - \phi_i.$$

Notice that $\phi_i = \#(\text{HoTs})$ which cancels the real cost. Remember that after a deletion, we consolidate our trees. We will have at most $O(\log n)$ roots. So, $\phi_f = O(\log n)$. We get amortized cost of $O(\log n)$ delete-min.

For decrease-key, just cut that subtree and put it in our collection of HoTs. The big problem is that we lose our binomial tree structure and cannot rely on our degree bounds.

§1.3 Analyzing the tree ("Saving Private Ryan")

If a node loses more than one child, we will cut that node and move to the collection of HoTs. We will implement this using a "mark bit" by setting it on first cut and clearing it on second cut. This might cause cascading cuts up the ancestor line until we find a cleared mark bit.

To analyze this situation, we must:

1. Show that cascading cuts cost free (amortized).
2. Show that tree sizes remain exponential in root degree. The base is unimportant. If degree d implies size c^n, a tree of size n implies degree is $O(\log_c n)$.

§1.3.1 Step 1: Analyze mark bits

Define ϕ as $\#(\text{mark bits})$. What is the cost of decrease-key?

- $r_i = \#(\text{cut nodes}) + 1$ where 1 is the housekeeping required to change key.
- $\Delta\phi = -(\#(\text{cut nodes}) - 1) + 1$ (+1 at the end for setting the top mark bit)
- $a_i = r_i + \Delta\phi = O(1)$

This analysis is **cheating**: we changed the ϕ!

Let's try again. Let $\phi = \#(\text{HoTs}) + 2 \cdot \#(\text{mark bits})$.

§1.3.2 Step 2: Understanding tree size

Let's say we have node x. Consider current children indexed from 1, ordered by when they were added: $y_1, \ldots, y_n$. Claim: y_i has degree $\geq i - 2$.

Proof. Consider y_i. This node was added after $y_1, \ldots, y_{i-1}$. When added, x had degree $\geq i - 1$. So, y_i had degree $\geq i - 1$. It is still in the tree, so that means only one child was cut from y_i. That means degree is $\geq i - 2$. □

Let S_k be the minimum number of descendants (including ourselves) of degree k node. Then: $S_0 = 1$, $S_1 = 2$, $S_k \geq \sum_{i=2}^{k} S_{i-2}$, meaning $S_k \geq \sum_{i=0}^{k-2} S_i$.

Solve at equality:

$$\begin{aligned} S_k &= \sum_{i=0}^{k-2} S_i \\ S_k - S_{k-1} &= S_{k-2} \\ S_k &= S_{k-1} + S_{k-2} \\ S_k &= \phi^k. \end{aligned}$$

We have proven that the degrees of root nodes are exponential.

§1.4 Summary of Fibonacci Heap

$O(1)$ insert, decrease key. $O(\log n)$ delete-min which is optimal. This bound is unavoidable because sorting problem in $O(n \log n)$ reduces to using priority queue. We can also merge in $O(1)$. (Will be covered later in the course.)

Is this practical? The constants aren't that bad, but we will have problems related to cache because Fibonacci heap uses pointers. Binary heap, considered an implicit heap, avoided this by storing everything in an array.

Remember that the original motivation for coming up with Fibonacci heap is Prim/Dijkstra. With regular heap, we have $O(m \log n)$ runtime. With Fibonacci heap, we will have $O(m + n \log n)$ because decrease-key operations now cost constant time.

§1.5 Fast Minimum Spanning Tree

Consider why finding minimum spanning tree is so slow. We were trying to do n delete-min operations on size n heap. Tarjan thus came up with an idea to keep the heap small. Here is how: choose parameter k. From a node, start running Prim. Insert neighbors and do delete-min as usual. Keep doing this until the heap size hits k. (Recall that the heap's contents are the neighbors of the current tree.) Then, we start Prim algorithm again at a new node **not** in the tree. Grow this new tree until we have k neighbors or join with existing trees. Eventually, all the nodes will be in a tree. Contract each tree into a node, and restart the MST algorithm with a new k.

What is the runtime for a phase with value k? As usual, we have at most $O(m)$ decrease-key operations in $O(1)$. Time complexity is $O(m+t \log k)$ where t is the number of nodes in a phase. If $t \log k = m$, then we have runtime of $O(m)$. So, set $k = 2^{\frac{m}{t}}$. This means each phase will take $O(m)$.

What is the number of phases? At the end of each phase, each node as k incident edges. That means #(vertices) $\leq \frac{m}{k}$, implying $t' \leq \frac{m}{k}$. $k' = 2^{\frac{m}{t'}} = 2^k$. Initially, $t = n$. Take $k = \frac{m}{n}$ or at least 2. Each time we run the algorithm, k exponentiates. Number of phases is the number of times we have to exponentiate $\frac{m}{n}$ to reach n. This is represented as $\beta(m, n) = \min \left\{ i : \log^{(i)} n \leq \frac{m}{n} \right\} = O(\log^* n)$.

Theoreticians are never satisfied. They managed to improve the runtime to $O(\log \beta(m, n))$ using edge bundles. Chazelle gets $O(m\alpha(n) \log \alpha(n)$ whee α is the inverse Ackerman function. If you do randomization, you get $O(m)$.

BUT! The weirdest thing of all about MST is that we actually have an **optimal** MST algorithm that we can't even prove the runtime.

§1.6 Persistent Data Structures

Sarnak & Tarjan introduced the idea of persistent data structures. We want to be able to answer questions like "What was the minimum value in the previous version?", basically querying into the past. This is called partial persistence. If you can insert a new key into the older past and ask the minimum value in the more recent past, this is called full persistence. (Basically time traveling!)

You can do this with any pointer-based data structures where each node has fixed size and has fields: old values, and pointers to other nodes. (No arrays!) Wrap data structures and primitive operations on it where we can change value or pointer in a node and query. So, that means we don't have to think about the whole data structure. Just think about **how to make a node persistent**.

Naive solution: change each field into an array of time-value pairs. This is called the "fat node" method. Maintain an array per field and add value on each field on modification queries in $O(1)$. Each query will take $O(\log t)$ because we can binary search the time t. This is a multiplicative slowdown which will hit every single operation we have in the data structure. Space is grown by $O(1)$ each time.

2 Persistent Data Structures

§2.1 Fat node and the copy method

One way we can create persistent data structures is to use **fat nodes**: replace each field by an array. Suppose we have done t updates thus far on a particular node. Each look up will take $O(\log n)$ time as we need to binary search the time array (multiplicative). Each modification takes only $O(1)$ as we append in the dynamic array. Note that we must also think about space complexity. Lookup is free. Modification takes $O(1)$ space per update, which is optimal.

Another way is to **copy the whole structure**. Each lookup takes additional $O(\log t)$ time, not multiplicative per node. Modification and space cost takes up to $O(n)$, though. This is huge.

§2.2 Path copying

Only a few nodes are ever changed, though. One way we can optimize is to only reconstruct the nodes that need to be changed and for the unchanged paths, use pointers to the old version. Lookup cost is still $O(\log t)$ additive. We cannot quite quantify time and space complexity of modification, though.

§2.3 Lazier path copying by Sarnak & Tarjan

Fat nodes are bad because they are too plump. The idea is to use some sort of hybrid: use plump nodes, but if they get too fat, resort to path copying. Sarnak & Tarjan came up with a way to create **partial** persistence on **tree data structures**.

In each node, add one extra timestamped field. Use it for the first modification of the node. On the second modification, make a copy with an extra field. Update parent's timestamped field to point to the new copy. (Basically, note that which pointer has to be overridden after what time.) If two children need to be changed (and thus do not fit into the extra field), we need to copy the root instead.

When we want to look up a field starting from a node at time t: First, check if the field was modified before time t. If so, use the extra field. This part takes $O(1)$ time (constant factor slowdown) but you still need an array of roots sorted by time, so $O(\log t)$ additive slowdown.

We will do amortized analysis to quantify space complexity. Notice that best case happens when all extra fields are empty. The worst case happens when we try to modify a node that already has its extra field filled up, and that only happens in the near present, not the old past. We define live nodes as all nodes reachable from current root. Define potential function: $\Phi =$ full live nodes. Let's consider two cases:

- Modifying when the extra field is empty: amortized cost is 1.
- Modifying a full node: do copy, point parent to a new copy, old node becomes dead (-1 to potential), so amortized cost is 0.

This means the cost of a full update (path copying) is $O(1)$.

§2.4 Application: Planar point location

Given a planar subdivision and a query point q. Find which polygon q is in. (n in these analyses is the number of segments.)

Consider the 1-D version: we are given a line and segment division. We can simply binary search to find on which segment q lies. Takes $O(\log n)$ time.

One of the trick that is used in computational geometry is **Dimensionality Reduction**. In this problem, take all the vertices of the planar subdivision, project them vertically. We simply need to figure out in which horizontal slab q is in $O(\log n)$, then later figure out where we are within that slab in 2-D using binary search which is also $O(\log n)$. The problem is to somehow preprocess the order of line segments for each slab.

Let's consider the complexity of building process. Space is number of vertices times number of segments, so $O(n^2)$.

Use persistent data structure instead. Sweep a vertical line from left to right and consider horizontal coordinate as "time" instead. At each vertex, some line segment crosses other line segments, appear, or disappear. We can use a persistent BST (e.g. red-black tree) to accommodate this. The cost for one modification is the number of primitive node changes, which comes from regular insertion, walking up and doing path rebalancing, recoloring, and rotations. It takes $O(\log n)$ time and space. The number of modifications is the number of segments, which is n, so total of $O(n \log n)$ cost.

We can refine this further. Notice that red black trees modify $\log n$ bits and do only one rotation. We never care about old red-black bits, so persist only the rotations. So, space goes down to $O(n)$.

Time to construct is $O(n \log n)$, and query time is $O(\log n)$. This is optimal, even in 1-D, because they map to sorting/binary searching.

3 Splay Trees

Splay trees are developed by Sleator & Tarjan. They are balanced binary search trees with $O(\log n)$ search, insert, delete. Some other examples are: red-black, AVL, scapegoat, 2-3, treap, etc. but they are annoying because they require adding extra info to maintain balance. Splay trees are **self-adjusting BSTs**. There is no balancing info, so splay trees actually end up unbalanced! Nevertheless, they outperform regular BSTs in formalizable ways. They also allow: merge, split, etc. in $O(\log n)$.

The analysis is sophisticated. They come up with potential function Φ that measures imbalance. They prove that it requires many inserts to cause imbalance, but the search decreases Φ to fix imbalance.

Let's start with the intuition. Notice that search is slow when the tree has a long path. Our goal would be to shorten it. We want to quantify this by having high potential for long paths. Each insert lengthens a path and increases the potential. Be aware that shortening our path would necessarily lengthen others.

Recall: rebalancing via rotations relies on keeping roughly the same subtree size in two children. Let's think about this differently: During search, if we descend to a smaller child, it is good. Our tree size is definitely halved, and can only be done so $O(\log n)$ times. If the tree is balanced, the larger tree also has $< \frac{2}{3}$ nodes, so the same logarithmic time bound also applies. Conversely, an unbalanced tree has some fat children, meaning we make almost no progress on each descent. Each fat child should have large potential.

Usually, we can use rotations to reduce a fat child but it might pull up just one node and leave a whole long path behind. In a splay tree, we solve this problem by using **double rotation**s (pick one depending on where x is relative to its grandparent).

Primitive operations are

Splay Double rotate x up until x is root (may require a single rotation at the end when x is root's child).

Search First, find x the usual way, then splay x up.

§3.1 Analysis

For simplicity, let's assume each item x has a weight w_x (just for analysis). For now, assume $w_x = 1$. Let's define $\text{size}(x)$ as the total weight of x's subtree, and $\text{rank}(x) = \lg \text{size}(x)$. rank indicates the fastest possible search we can do in this subtree. Let's define potential $\Phi = \sum_u \text{rank}(u)$. Notice that potential is high for bad trees. Each rotation which fixes the tree will decrease Φ.

Lemma 3.1.1 (Access Lemma)

Amortized time to splay(x) given root t is $3(r(t) - r(x)) = O\left(\log \frac{s(t)}{s(x)}\right)$. At the end x is the root, so its size is just the total tree weight (W). Therefore, the end rank $r'(x) = r(t)$. Amortized cost is $3\left(r'(x) - r(x)\right) + 1 = O\left(\text{change in } x \text{ rank}\right)$.

Consequently, if we have $w_x = 1$, $W = n$, that means $r(x) = \lg s(x) \geq \lg 1 \geq 0$. $r'(x) = \lg W = \lg n$. So, splay time is $O(\lg n)$. If we can prove this splay time bound, then find time bound is also restricted to logarithms (because find is always heavier than splay).

Proof. Show that amortized cost of a single double rotation is $3\left(r'(x) - r(x)\right)$. Notice that the sum telescopes: $3\left(r'(x) - r(x)\right) + 3\left(r''(x) - r'(x)\right) + \ldots = 3\left(r(W) - r(x)\right)$.

We will consider the zig-zig case only (which is harder). (Diagram is in iPad notes.) The real cost is 2 (double rotations). $\Delta\Phi = r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z)$. Note that $r'(x) \geq r(x)$, $r'(z) \leq r(z)$.

Assume $r'(y) \approxeq r(y)$. Suppose $r'(x) \gg r(x)$. Then, $3\left(r'(x) - r(x)\right) \gg r'(x) + r(x) + 2$. $\Delta\Phi$ absorbs the real cost.

Trouble happens when $r'(x) \approxeq r(x)$. This means A is fat. We're raising a fat tree. A is no longer child of y or z. y, z potentials will decrease a lot, so this will pay for the rotations instead.

Either way, change in potential absorbs the real cost. $\square$

Amortized cost $2 + \Delta\Phi = 2 + r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z)$. Notice, $r'(x)$ and $-r(z)$ cancels because z was the root of the subtree then x becomes the root. Now we have $2 + r'(y) + r'(z) - r(x) - r(y)$. Let's upper bound this quantity as we don't know much about y. We only know that $r'(y) \leq r'(x)$ as y is lower than x in the final tree. Originally, $r(y) \geq r(x)$ too. So, $2 + \Delta\Phi \leq 2 + r'(x) + r'(z) - r(x) - r(x) = 2 + (r'(z) - r(x)) + (r'(x) - r(x))$. It is sufficient to show that $2 + (r'(z) - r(x)) \leq 2\left(r'(x) - r(x)\right)$.

$$\begin{aligned} r'(z) + r(x) - 2r'(x) &\leq -2 \\ r'(z) - r'(x) + r(x) - r'(x) &\leq -2 \\ \lg \frac{s'(z)}{s'(x)} + \lg \frac{s(x)}{s'(x)} &\leq -2. \end{aligned}$$

Notice that the old $x - A - B$ subtree and the new $z - C - D$ subtree are completely disjoint. So, the last line could be written as

$$\lg \alpha + \lg\left(1 - \alpha\right) \leq -2$$

where the LHS is maximized when $\alpha = \frac{1}{2}$.

Now we know cost of splay is $\le 3(r'(x) - r(x))$. However, this is amortized cost, so we need to take a look at $\Phi_f - \Phi - i$ too. m operations have real cost $O(m \log n) - (\Phi_m - \Phi_0)$. How large can $\Phi_0 - \Phi_m$ be? Claim: $\Phi_0 \le n \log W$. $\Phi_m \ge \sum \lg w_x$. This means $\Phi_0 - \Phi_m \le \sum \frac{\lg W}{\lg w_x}$. Notice that this is basically the cost to splay each node once.

Corollary 3.1.2 (Balance theorem)

If $w_x = 1$ for each x, then amortized cost is $O(\log n)$.

Level k has 2^k spots. Any items with $p_i \ge 2^k$ can go in level k. Item x has depth $\log_2 \frac{1}{P_x}$. Overall search cost will be $\sum_x -mp_x \log p_x = m \sum_x -p_x \log p_x$.

Theorem 3.1.3 (Static optimality theorem)

Suppose we have a series of m operations where item x is accessed $p_m \cdot x$ times. The splay tree will achieve the best average search cost without knowing p_x.

Proof. Access lemma stated: $\log \left(\frac{W}{w_x}\right)$. Let's set $w_x = p_x$, so $W = 1$. Total cost would be $m \sum_x -p_x \log p_x$. □

Static finger Imagine setting a finger at a spot and start searching from that finger. Splay tree matches the best static finger.

Dynamic finger Splay trees also match that!

Conjecture 3.1.4 (Dynamic Optimality Conjecture)**.** Splay trees match the performance of the optimum dynamic binary search tree strategy.

4 Multi-level bucketing and van Emde Boas trees

§4.1 Dial's algorithm for SSSP

Consider shortest path problem. Usually, we have time complexity of $O(m + n \lg n)$ if we use Fibonacci heaps, or $O(m \lg n)$ for other heaps. If our edge weights are integers in $\{1, 2, \ldots, C\}$ with small C, we can imagine each edge of weight w as a chain of w edges and run BFS in $O(mC)$.

Look back at our priority queue solution. Notice that the values are integers and the queried minimum values are always non-decreasing. Dial came up with an algorithm to solve this problem quickly, which we will explore.

First, create an array of "buckets". Each bucket d has nodes with distance estimate d. The shortest path algorithm is obvious: scan the bucket from left to right, remove min when found, insert neighbors in buckets (to the right), keep going. During the course of algorithm, neighbor updates take $O(m)$, and scans take $O(D)$ where D is the maximum distance. Overall runtime is $O(m + D) = O(m + nC)$. Auxiliary space complexity is $O(nC)$.

§4.2 Tries

To improve, observe that at any time, nonempty buckets range from current distance d to $d + C$. This means we can store our array in a circle of size C (roughly). Space complexity is thus $O(C)$.

Suppose we create another "summary" bucket which divides up the original array into blocks of size b. Track the nonempty buckets in our summary. On each scan, traverse blocks till we find a nonempty block and then traverse than nonempty bucket. Each scan will take $O\left(\frac{nC}{b}\right) + nb$ total. Set $b = \sqrt{C}$.

We can do even better with three layers instead of two layers. Each lower bucket takes care of $C^{\frac{1}{3}}$ elements. Each higher bucket takes care of $C^{\frac{2}{3}}$. However, notice that if we try to use too many layers, changing the summary will take too much time. To be exact, time complexity of Dial's algorithm would be $O\left(mk + nkC^{\frac{1}{k}}\right)$.

We define a trie: A trie is a depth-k tree over arrays of size Δ (degree Δ trees). Consider a range of C possible values (generally stored in circular array).

$$\begin{aligned} C &= \Delta^k \\ \Delta &= (C+1)^{\frac{1}{k}} && \text{ideally power of 2.} \end{aligned}$$

Insert takes $O(k)$. Delete-min needs to find non-empty children of roots in each layer, so takes $O(k\Delta)$ time. Back to Dial's algorithm, overall runtime would be $O\left(mk + knC^{\frac{1}{k}}\right)$. We can balance this by setting $m = nC^{\frac{1}{k}} \to k = \log_{\frac{m}{n}} C$. Overall runtime ends up being $O\left(m \log_{\frac{m}{n}} C\right)$

§4.2.1 Denardo & Fox

Right now, our insert/delete takes $O(k)$ time, and delete-min takes $O(\Delta k)$ time. Denardo & Fox ('79) realizes we don't have to waste time expanding our k level of buckets until we actually reach the point where we need to distinguish the order between two items in the same bucket. Basically, we will have only one "active" trie node per level on path to current minimum. Other items stay in buckets of lowest active node they belong in. Keep the counts of items in each layer (not counting descendants). Only expand the buckets when they become the minimum.

To insert, start at the top, and walk down until I land in a non-minimum bucket at some level, incrementing that level's item count.

To decrease-key, we **might** have to remove from current bucket. The new bucket is either behind current item on same level or below, so we descend from current to new level/bucket. (Can't be in front because non-decreasing property of shortest path.)

To delete-min, remove current min up to first non-empty layer. Scan for non-empty bucket (up or right), then expand down to create new bottom layer & new min. The last part's cost is already accounted for by cost of k per item moving down.

We have scan in Δ, insert in k and decrease key in 1. Our shortest path algorithm would take $O\left(m + n\left(k + \Delta\right)\right)$. If balanced, we get $O\left(m\left(\frac{\log C}{\log\log C}\right)\right)$.

§4.2.2 Cherkassky, Goldberg, Silverstein

Use D & F queue but add a standard heap to find the minimum entry at the top level. Make the top level much bigger than other levels so using a heap is worth it. This allows Dijkstra's algorithm to run in $O\left(m + n(\log C)^{\frac{1}{3}}\right)$. This is called "HoT queue" (heap on top of queue).

David Karger suggests we look at Goldberg's papers so we have an idea on how to research Experimental Algorithms.

§4.3 van Emde Boas Tree (1977)

vEB recognized that finding a non-empty bucket is a priority queue problem, so he goes all into recursion. For now, instead of talking about integers, we will talk about b-bit words, meaning $C = 2^b$.

The structure of vEB for b bits, which we will call Q, stores:

- Q.min: the minimum value in the queue.
- An array Q.low of size $\sqrt{C}$ ($\frac{b}{2}$ bits) where each index stores a vEB queue on $\sqrt{C}$ values in range.
- Another vEB called $Q.high$ of nonempty blocks. Notice that we store up to $\sqrt{C}$ numbers, each with $\frac{b}{2}$ bits.

§4.3.1 Insert(x, Q)

If $x < Q.min$, swap them. Then, put x into the recursive structure. Let's think of x as (x_h, x_l) where x_h is the $\frac{b}{2}$ high bits and x_l is the $\frac{b}{2}$ low bits. We need to check if $Q.low[x_h]$ is non-empty. If yes, then just insert x_1 into $Q.low[x_h]$, which is simple. Otherwise, create a vEB queue for $Q.low[x_h]$ and insert x_l into it. You also have to insert x_h into $Q.high$ to denote that that block is now non-empty.

$$\begin{aligned} T(b) &= O(1) + T\left(\frac{b}{2}\right) \\ &= O(\log b) \\ &= O(\log \log C) \end{aligned}$$

§4.3.2 Delete-min(x, Q)

First, find the min. Remove it. Then, replace $Q.min$ with the min of the recursive structure, which can be done by looking at $Q.high.min$ (gives x_h). If $Q.high.min$ is null, that means the recursive structure is empty, which means we should just set $Q.min$ to be null. Otherwise, look in $Q.low[x_h]$ and delete x_l. Then, $Q.min := (x_h, x_l)$. Beware, if deleting x_l gives empty block, then also delete x_h from $Q.high$. (This only happens if the first delete-min has only one item.)

Runtime is $O(\log \log C)$.

§4.3.3 Analyzing the space complexity

Suppose we want to create a 64-bit queue. If you insert only one item, we need 2^{32} slots at the top level and 2^{32} per item. Recall we use the arrays for quick $O(1)$ lookup. Why not use hash tables instead?

5 Hashing

We want to create a dictionary data structure that supports Insert, Delete, and Lookup on keys.

Model: our keys are integers in the universe $M = \{1, \dots, m\}$. If we want to store only n keys, can we use n space instead of m and still get $O(1)$ time? Phrased weaker: can we use s space where $s > n$? The idea is to use a hash function $M \to (S = \{1, \dots, s\})$ and store key k in array at position $h(k)$.

There is a problem of collision. Correctness isn't a big of a problem as we can solve it easily via chaining. Operation time does, however, grow linearly if everything hashes to the same index.

For all hash functions, there exists a large set of keys that collide badly (created by an adversary). The solution is to create a **hash family** which is a set of hash functions such that we can pick a good one for any set of items. Picking can be done via randomization. We analyze the average number of collisions with each item. We define for a pair of item i, j:

$$C_{ij} = \begin{cases} 1 & \text{if items } i, j \text{ collide} \\ 0 & \text{otherwise} \end{cases}$$

. Expected time to find is $\mathbb{E}[1 + \sum_j C_{ij}] = 1 + \sum E[C_{ij} = 1 + \sum \Pr[i, j \text{ collide}] = 1 + \frac{n}{s} + 1 + \alpha$.

The problem is remembering the random h we choose. Naively, it takes $m \log s$ space as each hash function can be written in form of how each value in M maps to indices in S.

§5.1 2-universal hash family by Carter and Wegman

For our collision analysis, we considered colliding pairs. We did not consider the entire sequence of operations. This means it is sufficient for h to be **pairwise independent**: it looks random to each pair of values even though it might not be truly random (**mutually independent**). Why is this useful? Suppose we deal with 3 coin flips x, y, z, but we only flip x and y and set $z = x \oplus y$. We can say $\Pr[x = y]$, $\Pr[x = z]$, $\Pr[y = z]$ are $\frac{1}{2}$ despite x, y, z not being mutually independent.

To generalize, suppose s is a prime number p. Carter-Wegman maps x to $(ax + b) \bmod p$. If a, b are chosen at random from $\{0, \dots, p-1\}$ then $h_{ab}(x)$ and $h_{ab}(y)$ are uniform over $0, \dots, p-1$ and pairwise independent.

To analyze, we want to compute $\Pr[h_{ab}(x) = s \wedge h_{ab}(y) = t \mid x \neq y]$. For mutually independent cases, this should be equal to $\frac{1}{p}$.

For pairwise independent, let's see $\Pr[ax + b = s \wedge ay + b = t]$. We can write this using matrix notation:

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

which has a unique solution only if $x \neq y$ which only comes from unique (a, b) that happens to work. The probability is $\frac{1}{p^2}$.

If s is not a prime, we choose $p \gg s$. Now, $h_{ab}(x) = ((ax + b) \bmod p) \bmod s$. There is some off-by-one imbalance though because p is prime. The probability of falling into each bucket differs by about $\frac{s}{p}$. If $p = n \cdot s$ then probability differs by $\frac{1}{n}$. This only changes the expected collisions by $O(1)$.

We can prove that the maximum load is $O(\sqrt{n})$ if we map n items into $n = s$ universe. This happens with probability $1 - \frac{1}{n}$. Some items will have larger load but they are unlikely to be sought after. For fun, theoreticians have also proven that there exist pairwise hash families where $\sqrt{n}$ max load is very likely.

§5.2 Guaranteeing $O(1)$ lookups

Suppose we already given the keys in advance. In fact, let's be ambitious and aim for no collisions at all. Obviously, $s \geq n$.

What is the chance of having no collisions if we pick a random h?

$$\begin{aligned} \mathbb{E}[\text{total \# of collisions}] &= \mathbb{E}[\sum_{i<j} C_{ij}] \\ &= \sum_{i<j} \mathbb{E}[C_{ij}] \\ &= \sum \Pr[i, j \text{ collide}] \\ &= \sum_{i<j} \frac{1}{s} \\ &\approx \frac{n^2}{2s}. \end{aligned}$$

If $s \geq n^2$ then we get $\leq \frac{1}{2}$ expected collisions.

Theorem 5.2.1 (Markov's Inequality)
If $X \geq 0$, then $\Pr[x > t] \leq \mathbb{E}[x]/t$.

In our case, $\Pr[\#(\text{collisions}) \geq 1] \leq \frac{1}{2}$.

Algorithm: let's try a random hash function. If it is perfect, we are done. It it fails, just try again. This is guaranteed to find a perfect hash function. On average, this takes only two trials, each trial taking $O(n)$ time.

This is still bad because we need $O(n^2)$ space for n items. We intuit that this is because the $\frac{n^2}{2s}$ bound is not tight.

§5.3 Fredman, Kombs, Szemeredi

They introduce the idea of 2-level hashing. First, you hash n items to $O(n)$ space. This will cause some collisions but not too many. Let's build a perfect hash table on each bucket (If there is b items, use b^2 space.). Intuitively, this takes $O(1)$ lookup time.

What is the total space? Let b_k be the size of bucket k.

$$\begin{aligned} S &= n + \sum_k (b_k)^2 \\ &= n + \sum_k \left(\sum_i \operatorname{Ind}[i \in b_k] \right)^2 \\ &= n + \sum_k \left(\sum_{i,j} [i \in b_k][j \in b_k] \right) \\ &= n + \sum C_{ij} \end{aligned}$$

.

We know:

$$\begin{aligned} \mathbb{E}[\sum C_{ij}] &= \mathbb{E}[\sum_i C_{ii}] + 2\mathbb{E}[\sum_{i<j} C_{ij}] \\ &\leq n + \frac{n^2}{2s}. \end{aligned}$$

If $s = O(n)$, $\mathbb{E}[\sum (b_k)^2] = O(n)$.

Finally, to guarantee that this happens, do Markov again. If expected space is $\leq cn$, then generate table till we get space $< 2cn$. (If we go through our lectures, we have c of about 13. If we do a bit more work, we can get c down to 6. In the paper, they can do $1 + o(1)$ using some techniques.)

6 Maximum flow problem

Combinatorial optimization problems are problems where you have to compute optimal output satisfying the constraints of "feasible" solutions. Each solution has a cost or a value, and you want to find the feasible solution of optimal cost/value. Steps to solve:

1. Understand feasibility
2. Construct an algorithm to determine feasibility
3. Find any feasible solution
4. Verify optimality of feasible solution
5. Compute optimal solution

The maximum flow problem is as follows: Given a directed graph G of m edges and n vertices. Given a designated "source" vertex s and "sink" vertex t. Along each edge e we have capacity u_e. We want to find flow for each edge: f_e. A solution is feasible on these conditions:

- Capacity: $0 \leq f_e \leq u_e$
- Conservation: $\forall v, \sum_w f(v,w) - f(w,v) = 0$ except $v \in \{s,t\}$.

Value of the flow network is $\sum_v f(s,v) - f(v,s) = |F|$. Maximize $|F|$.

Define "gross flow". Net flow on $e = (v \to w)$ is $g(v,w) = f(v,w) - f(w,v)$. This allows the conditions to be written a bit differently:

- Skew symmetry: $g(v,w) = -g(w,v)$.
- Conservation: $\sum_w g(v,w) = 0$.
- Capacity: $g(e) \leq u(e)$.

Value is $\sum_v g(s,v)$.

§6.1 Feasible Flow

Suppose we have a graph with vertices s and t. These are some of the obvious feasible flows:

- 0 flow

- A path from s to t where flow $\leq$ min capacity

Lemma 6.1.1 (Path Decomposition)
Any s-t flow can be decomposed as a sum of $s - t$ paths and cycles (with amounts).

Proof. Proof by induction on the number of edges carrying nonzero flow. Given a nonzero flow, we will find a path, then remove it. If there is a nonzero flow, that means there is an edge leaving s carrying the flow to v. There is also an edge leaving v carrying flow because of conservation. We can keep going until $v = t$ or we reach a previously visited vertex (enter a cycle). Find min flow edge on that path/cycle then decrease flow on each edge by that much. What remains is still a flow because we subtracted the same amount of flow entering or leaving each vertex. Each step zeroes a flow on at least one min edge. This can only happens m times, so this induction terminates. (This means a flow can be decomposed into $\leq m$ path flows.) $\square$

Is there a nonzero feasible flow in G? Yes, when t is reachable from s (by positive capacity edges). Let us prove this statement in both directions:

- If t is reachable from s, there exists a s-t path along which we can put a flow through.
- If there is a nonzero feasible flow, we can decompose into paths and cycles. The caveat is that if our decomposition is all cycles, then net flow is 0, which contradicts the assumption, so we can ignore this case.

Suppose there is no feasible flow, how can we show a certificate of this no-instance? Consider the set S of vertices reachable from s. $T = V - S$. By construction, there is no edge from S to T. We call (S, T) a **cut with no crossing edges**.

§6.2 Maximum flow upper bound

How do we verify that a given flow is maximum? We can try to find an upper bound and show that it is tight.

Consider a cut (S, T). Claim: maximum flow is less than or equal to the capacity of cut edges. Decompose flow into s-t path. Each flow has to cross this (S, T) cut and thus uses up some capacity on a cut edge. If max flow is less than or equal to the capacity of any cut, then it is less than or equal to the minimum possible (S, T) cut. (Hint hint, min-cut max-flow!)

Suppose a flow is currently not maximum. We can make it better by add flow on some of the available paths. However, there might be zero such paths. To solve this, we define a

residual graph. Given flow f, residual capacity $u_f(v,w) = u(v,w) - f(v,w)$. We know $u_f(w,v) = u(w,v) + f(v,w)$, so it is as if we can "undo" flows on certain edges.

If there exists a path in the residual graph, we can "augment" the flow. Now, we prove that if there is no augmenting path in residual graph, we have a max flow.

Proof. There is no s-t path in G_f. This means there is a s-t cut $(S, \overline{S})$ with no residual edge. Every $(S, \overline{S})$ edge in G is "saturated". Every $(\overline{S}, S)$ edge in G needs to be empty or else that would imply reachability. Now, let's decompose the flow into paths. Each path crosses the cut and uses up some capacity exactly once. Sum of paths is equal to $(S, \overline{S})$ capacity. So, we have shown that the value of flow is equal to the value of cut. This flow is maximum. The cut is also minimum. □

Theorem 6.2.1 (Max Flow Min Cut Theorem)
These are equivalent:

- f is max flow.
- There is no augmenting path in G_f.
- Value of f is equal to capacity of $(S, \overline{S})$ for some S.

Proof. If there is an augmenting path, we can increase f, so the flow is not maximum. If there is none, then let S be reachable vertices, all edges leaving S is full and entering is empty. So, the value of flow is the capacity of the cut. □

Corollary 6.2.2
Net flow across any cut is equal to the flow value.

Problem 6.2.3. To build intuition for this, try constructing

1. A graph with one max flow and one min cut.
2. A graph with one max flow but multiple min cut.
3. A graph with one min cut but multiple max flow.
4. A graph with multiple min cut and multiple max flow.

§6.3 Ford-Fulkerson Algorithm

If flow is not maximum, increase it with an augmenting path in G_f. Keep doing this until it terminates.

In one iteration, run BFS in G_f, taking $O(m)$ time. Assume integer capacities, the number of iterations is at most the maximum flow value. Total runtime bound is $O(mf)$. f is bounded by the $m\mu$ where μ is the maximum edge.

This is a pretty bad runtime if capacities grow big. This is even worse when we are dealing with rational values because the product of all denominators can blow up. If we are dealing with real number, it might take infinite amount of time to converge to a non-maximum flow (very bad!).

Corollary 6.3.1 (Max flow intergrality)
If G has integer capacities, then there exists an integral max flow.

§6.3.1 Finding better augmenting paths

One idea is to find an augmenting path which increases the flow by as most as possible: maximize the bottleneck capacity. Finding the bottleneck can be done by sorting the edges and then binary search for bottleneck, testing if there is still an s-t path. This takes $O(m \log n)$. We can do faster by doing Prim's algorithm (directed MST) in $O(m+n\log n)$ using Fibonacci heap.

Does this really help? There might be a case where any augmenting paths have very low (bad) bottlenecks. Let us try to figure out how many augmentations are required to reach maximum flow.

From path decomposition, we know that any flow uses at most m paths. This implies that at least one of these paths has $\frac{f}{m}$ flow (f in residual). So, the maximum bottleneck possible has at least $\frac{f}{m}$ flow. One augmentation takes our desired flow f to $f - \frac{f}{m}$ remaining flow. After k iterations will take us from f to $f\left(1-\frac{1}{m}\right)^k \leq fe^{-\frac{k}{m}}$ because $1-x \leq e^x$. So, if $k \geq m \log f$, required flow would be less than 1. For integral capacities, finding $m \log f$ maximum augmenting paths gives us max flow. Runtime is $O\left(m^2 \log f\right) = O\left(m^2 \log mu\right) = \tilde{O}\left(m^2 \log u\right)$. This is polynomial even if we are using rational capacities!

§6.3.2 Scaling algorithm (Gabow '85, Dinitz '73)

Apply the idea from "unit case" of the problem to bigger numbers. Remember that our generic augmenting path algorithm takes $O(m^2)$ time on unit capacity graphs. Key insight: numbers are just sequences of bits. Each bit is a unit case. So, it's like we are rounding backwards: first, start with optimal solution to a rounded problem (considering the large bits only), and then try to fix up solutions to "unround".

If we are given a max flow for current shift level and we shift another bit, all capacities are doubled and some are incremented by one. How to find a new max flow? First, double previous max flow. Min-cut is still saturated now, so residual graph has 0 S-T capacity. Now, as we increment by one, the number of augmenting paths is at most m. Each augmenting path takes $O(m^2)$ time. So, total runtine is $O(m^2 \log u)$.

§6.4 Strongly Polynomial Max Flow Algorithms

Is it possible to have an algorithm that runs independently of the number sizes, assuming we can add and subtract the numbers in $O(1)$ time? This is called "strongly polynomial." (Previous scaling algorithms are called "weakly polynomial.")

§6.4.1 Edmonds-Karp Algorithm

We need a different measure of progress that is not just the value of flow. We know that we have reached max flow when there is no path from s to t, so can we quantify the notion of "almost cutting path from s to t" somehow? Let's say we have a cut when there is "infinite" distance from s to t. Large distance implies nearly max flow.

This implies obviously that we should try to pick the shortest path to do augmentations (SAP: Shortest Augmenting Path). This can be found in $O(m)$ via BFS. Claim: we need at most $O(mn)$ SAPs, so max flow can be found in $O(m^2 n)$ time. This is slower than scaling, however.

Lemma 6.4.1

Let $d(v)$ = distance from s to v. If you augment on a SAP, then $d(v)$ do not increase.

Proof. Proof by contradiction.
Suppose that some distances decrease. Let $d'(v)$ be the distance after augmentation that $d'(v) < d(v)$. Let v be such vertex with minimum $d'(v)$. (Obviously, $v \neq s$.) Let P' be the shortest path to v after augmentation. Let w be the vertex preceding v on P'. (It is possible that $w = s$.) Note that $d'(v) = d'(w) + 1$. However, since $d'(v)$ is the minimum, we have $d'(w) \geq d(w)$. (Otherwise, w would have been the smallest.)
We claim that (w, v) is "new", meaning it has 0 capacity before SAP. Otherwise, in prior graph, path to w then $w \to v$ is a path to v with length $d(w) + 1 \leq d'(w) + 1$ and would imply $d(v) \leq d'(v)$ which contradicts the claim. The only way a new edge can be created when SAP sends flow on reverse edge (v, w). This means (v, w) was on the SAP,

which implies $d(w) = d(v) + 1 \implies d(v) = d(w) - 1 \leq d'(w) - 1 = d'(v) - 2 < d'(v)$. Contradiction. □

Now that we have proven that distances do not decrease, we have to show that they eventually increase.

Lemma 6.4.2

After $\frac{mn}{2}$ SAP, we have a max flow.

Proof. Consider some SAP that saturates the edge (u, v). (u, v) is now gone from the graph but it might return later when some later SAPs use (v, u). When we use (u, v), $d(v) = d(u) + 1$. When we use (v, u), $d'(u) = d'(v) + 1 \geq d(v) + 1 = d(u) + 2$. This means the next time we use (u, v), $d(u)$ will have increased by 2. After $\frac{n}{2}$ saturations of (u, v), $d(u) \geq n$, meaning u is unreachable and thus (u, v) cannot be used again. Each edge can be saturated at most $\frac{n}{2}$ times. Since each SAP saturates at least one edge, it takes at most $\frac{mn}{2}$ to give us a max flow. (Note that m here includes backward edges.) □

Final runtime for our max flow algorithm is $O(m^2 n)$. This is strongly polynomial.

§6.4.2 Blocking Flows

Note that when we run BFS and create a "layered graph", the shortest paths in it stay shortest after we perform SAP (because shortest paths never decrease).

Definition 6.4.3 (Admissible edge)**.** An admissible edge is an edge that goes forward one layer.

Definition 6.4.4 (Admissible path)**.** An admissible path is a path where all edges are admissible.

Definition 6.4.5 (Admissible graph)**.** An admissible graph is a graph where we consider only the admissible edges.

Claim 6.4.6 — An admissible path is a shortest path.

Let us consider what kinds of edges can be non-admissible. They might go backwards, stay in the same layer, or go forward by greater than one layer. (An admissible graph is basically a shortest path tree.)

The idea is to build an admissible graph, and use it to destroy **all** shortest paths, then repeat with a new admissible graph.

Definition 6.4.7 (Blocking flow)**.** A blocking flow is a flow using only admissible edges that saturates an edge on every admissible path.

After finding a blocking flow, there will be no admissible paths left in the residual graph. This means any (s, t) path after this will use non-admissible edges. Shortest path must definitely increase, so after n rounds we will find a max flow.

Now, we consider the problem: Given an admissible graph, find a blocking flow.

Bounds for unit capacity graphs

One general insight when dealing with graph or flow problems is to consider the unit capacity case. Suppose we start at s and advance along some admissible edge to v. If $v \neq t$, keep advancing one vertex at a time until there are no more admissible edges or we reach t. If we reach t, just saturate them (remove all edges) and start again. If we cannot go any further, return back and mark v as useless. Keep doing until we are blocked at s. Runtime is $O(m)$ because the number of retreats does not exceed $O(n)$, the number of augmentations does not exceed $O(m)$, and the number of advances is the number of retreats plus number of augments.

For unit capacity graph, our total runtime of max flow would be $O(nm)$. This does not seem very exciting, but remember that we do not require simple edge assumptions for this graph.

There is a better bound for unit capacity. Suppose we find k blocking flows. That means $d(s, t) \geq k$. By flow decomposition, there are at most $\frac{m}{k}$ destroying paths because each path uses up k edges and each edge has unit capacity. Since each blocking flow will find at least one unit of flow, we only need other $\frac{m}{k}$ additional blocking flows will finish the algorithm. Runtime is $O\left(km + \left(\frac{m}{k}\right) m\right)$. Set $k = \sqrt{m}$, so runtime is $O\left(m^{\frac{3}{2}}\right)$.

We can also prove that the runtime is $O\left(mn^{\frac{2}{3}}\right)$ for simple graphs. The runtime is $O(m\sqrt{n})$ in bipartite graphs.

Generalizing to other graphs

If we start at s and advance until we find $v = t$ and saturate or get blocked and retreat. There are at most m retreats, still. The number of advances does not matter because we will either augment a path or retreat. Now, we consider how many augmentations may happen. We cannot carry over the proof from previous section because each augment only destroys one edge, not all, so there might be at most m augments in a round.

We can improve the bound by using amortized analysis. Charge each augmentation to one edge. We might need n work per edge. This means each blocking flow takes mn

time. Max flow algorithm will take $O(mn^2)$ time which is somewhat disappointing but is better than the non-blocking-flow algorithm!

To refine: If the value of max flow is f, then augments only cost nf total (summing over many blocking flows). Other work is $O(m)$ per blocking flow due to retreating. So, max flow runs in $O(mn + nf)$.

Remember that we can use the scaling technique to make sure the f term is small term is small. Recall that at most m residual flow is present because we had an empty cut and each crossing edge gains at most one unit of capacity at most. So, blocking flow runs in $O(mn + nf) = O(mn)$. Total runtime is thus $O(mn \log u)$.

7 Minimum-cost Flow Problem

§7.1 Min cost max flow and variants

This problem is similar to standard max flow problem but each edge e has per-unit-flow cost $c(e)$ associated with it. We want to find a max flow f that minimizes $\sum_e c(e)f(e)$. Note that $c(e)$ may be negative. (Now you know why we don't use c for capacity up until now. c is reserved for "cost." Some people may ask why we don't use p for price instead, but it turns out price is also reserved for something else.)

Pushing flow through an edge (u, v) creates a residual edge (v, u). How should we define the cost on the reverse edge? The cost should be $-c(u, v)$ because pushing flow back should result in a refund.

A variant of this problem is to find min cost to send v units of flow. This can be solved easily by capping the max flow via a bottleneck edge. This generalizes the max flow problem but also generalizes the shortest path problem! To send a flow of one unit from s to t we can just find the shortest path under $c(e)$ metric.

Path decomposition still applies to min cost max flow: Any flow is a sum of paths and cycles. The cost of the flow is the sum of the per-unit cost of the paths/cycles times the capacity of the paths/cycles.

Another variation of this problem is called **min cost circulation**. We have a **circulation** which is a flow with balance everywhere and no source or sink. This problem becomes interesting when some costs can be negative because it might make more sense to send a circulation rather than none to gain some profit. For this problem, cleaner decomposition claim applies: we can decompose a circulation into at most m cycles.

Problem 7.1.1. Reduce min cost circulation (MCC) to min cost max flow (MCF). (We are given a black-box for solving min cost max flow. Find a way to solve min cost circulation.)

Solution. Just create source s and sink t outside the graph and we're done because max flow is always 0 but min-cost max flow always tries to minimize min cost with cycles. □

Problem 7.1.2. Reduce min cost max flow to min cost circulation.

Solution. Solution: Create an edge from t to s with infinite capacity and negative infinity cost. This encourages the circulation to go through (t, s), meaning other edges form a flow from s to t. Of course, we can't really write infinite capacity, so we can write mU instead which is the upper bound on the max flow. For the cost, write $-mC$. (U = max capacity. C = max cost.) In fact, $-nC$ is sufficient because each flow goes through at most n nodes. (It's actually nCU but don't forget that this is per-unit cost.) □

Alternate solution.

1. Find a max flow f.
2. Want min cost flow f^*. The difference $f^* - f$ is a circulation but may not be feasible because of possible negative flows. Yet, we claim that it is feasible in $G_f = G - f$. Check: if $f_e^* - f_e \geq 0$, there is a positive flow on e with amount $\leq u_e - f_e = u_f(e)$ because f^* is feasible by definition and fits in G_f. If $f_e^* - f_e \leq 0$, there is negative flow on e which is the same thing as a positive flow of $f_e - f_e^*$ on reverse of e, and this is feasible because G_f by definition has capacity f_e on reverse of e.
3. Flow per edge: cost of circulation is $C(f) - C(f^*)$. Claim $f* = f + q$ where q is the min cost circulation. (First, find the max flow, then somehow reduce it.)

□

§7.2 Showing optimality

Given f, is it a min cost flow, assuming we can already verify that it is a max flow?

Claim 7.2.1 — f is MCF if and only if G_f has MCC with cost $= 0$.

Proof. If there is a MCC with cost < 0 then we would have been able to get a better flow. To prove in other direction by contradiction, suppose that there is a negative MCC in G_f. By decomposition into cycles, at least one of the cycle must be negative. (If there is a negative cycle, then there is a negative circulation: just take that cycle only.) But if there is a negative cycle in G_f then we can just add it to improve cost. □

The mechanism for verifying MCF is to simply check if there is a negative cycle in G_f. This can be done via Bellman-Ford's algorithm for shortest path.

Bellman-Ford discovered that the shortest path is not defined because of a negative cycle and so we can identify that cycle easily. That provides us the certificate of MCF. Thus, verification for MCF can be done in $O(mn)$.

§7.2.1 Cycle canceling algorithm (Klein)

Given current f, find a negative cycle in G_f and send flow through it capped at the bottleneck. This saturates the cycle.

Each cycle drops cost of f by at least 1. Possible circulation costs are in range $[-mCU, mCU]$ (as proved earlier in MCC). So, $O(mCU)$ cancellation suffice. Therefore, we have a pseudo-polynomial $O(m^2nCU)$ time algorithm.

To be fair, cycle canceling is not a bad idea. It just has to be applied well, like with augmenting paths. We'll explore these ideas in the homework instead.

§7.3 Prices and reduced costs

Let's think in term of markets.

Even though there's only demand at source s and sink t, we can think of this problem as if we have "merchants" at each node. If we give away the flow for free from s, then merchants will get it and sell to neighboring cities by negotiating the prices $p(v)$ associated with each node v to earn some money, paying for transport costs in the process.

What determines prices? When is a set of prices plausible/stable? Let's look locally at v and w. There's a profit opportunity only when $p(v) + c(v, w) < p(w)$: merchant buys things at v, ship it to w, and sell it at a higher price. We define the reduced cost $c_p(v, w) = c(v, w) + p(v) - p(w)$ so we will have profit when $c_p(v, w) \le 0$. This represents the "net cost" for moving things from v to w.

What will happen? Demand at v will increase (because people want to buy there to sell for profit) and decrease at w (because people don't want to buy there). This change will happen until the reduced cost is positive or edge (v, w) is saturated. So, we define feasibility of a price function as when the market is stable.

Definition 7.3.1. Price function is feasible for (residual) graph if and only if no (residual) arc has negative reduced cost.

Important observation: prices don't change cycle costs! On any cycle, $\sum c_p(v, w) = \sum c(v, w)$. You can observe this by writing the terms out and the terms cancel/telescope. No matter what price function you use, the negative cycles are always the same ones.

Lemma 7.3.2
A circulation/flow is optimal if and only if there exists a feasible price function.

Proof. First, we prove that if we have a feasible price function p, this implies optimality, which means we have no negative residual cycle. Remember that the definition of feasible p is that we have no negative reduced cost edges. Then, there are no negative reduced cost residual cycles. By previous observation, there is no negative cost residual cycles (non-reduced). So, flow is optimal.

Now, suppose we have optimality and want to show that there exists a feasible price function p. We have no negative cost cycles, and thus no negative reduced cost cycles. Idea: $p =$ shipping cost from s. We attach a vertex s' with 0 cost edges to every vertex. Compute shortest paths from s' to all v. Set $p(v) =$ shortest path cost. This works because from optimality there's no negative cycles. p is feasible because triangle inequality holds for shortest path and our definition of price is basically that.

(We used s' instead of s because we want to ensure that every vertex has a price. 0 cost edges work because in min cost flow, the cost is either 0 or negative.) □

Now, observe that increasing $p(v)$ makes $c_p(u, v)$ smaller and $c_p(v, w)$ larger. Ideally, we can do this kind of adjustment until all edges are positive.

§7.3.1 Scaling algorithms using prices

Before: we start with a max flow f and then find the MCC q by cycle canceling to get $f^* = f + q$. Now: we start with an empty flow which is already at min cost, and now we do augmenting paths to make max flow but make sure that we have min cost at all times.

Which augmenting path should we pick? Try to pick the shortest augmenting path under cost metric.

Claim 7.3.3 — Suppose we have no negative cycles in residual graph G_f. Then, SAP will not create one.

Proof. Proof by contradiction. Suppose SAP creates a negative cycle. This means we've created an edge that constitutes the negative cycle. The cycle intersects our shortest path from s to t somewhere. So, consider new path: original path plus the negative cycle. The sum of costs would be cheaper because we're adding a negative cycle. Therefore, SAP isn't actually the shortest. □

Alternate proof. Notice that price function shifts all s-t path costs by the same amount: $p(t) - p(s)$. (Same argument that cycles are unchanged.) During SAP, residual graph never has a negative cost cycle. Proof by induction. Suppose there are no current negative cycles. Then, we can compute shortest paths from s. Use these shortest paths to define new price $p(v)$. Then for all edges (v, w) on the shortest path, $p(v) + c(v, w) = p(w)$ because we're on a shortest path, and the reduced cost is thus $0 = c_p(v, w) = -c_p(w, v)$. This means our augmentation only introduces zero cost edges and price function stays feasible. □

The algorithm is then obvious: start with no negative costs, keep finding SAP, then once we get max flow we will also have MCF because price function is always feasible.

For unit capacities, each augmentation finds a unit of flow, so max flow is done after $f \leq mC$ iterations. Each round takes $\tilde{O}(m)$, so runtime is $\tilde{O}(mf)$. Caveat: Dijkstra's algorithm requires non-negative edges but that's okay because we can keep updating our feasible price function and use the reduced costs without negative values.

This algorithm has two limitations: we assumed at the beginning that we have no negative arcs and that we have unit capacities.

Unit capacities can be dealt with easily with scaling: Shift in one bit of capacities at a time. May create residual flow. We have to eliminate with more augmentations then we'll finish after $O(\log f)$ shifts. Caveat: can shifting in a bit create negative cost arcs? On shifting, we double capacities, double the flows, and maybe add 1 unit capacity on each edge. Maybe a 0-capacity edge becomes 1-capacity edge and it can be a problem if the cost is negative.

Let's find a way to deal with negative arcs, assuming capacity 1.

1. Send flow on all negative arcs (v, w). This will create a residual edge with positive cost but violates conservation, so declare deficit on v and excess on w.

2. Send excesses to deficits by cheapest possible routes. This is a min cost flow problem but with no negative arcs, so we can compute a fast min cost flow using Dijktsra's algorithm and get feasible price functions for MCF/circulation such that the residual graph has no negative arcs.

This is an algorithm that finds MCC.

Back to our scaling algorithm, we just have to saturate the negative arcs and use MCF to balance whenever there's a problem. Runtime of this step is $\tilde{O}(mf)$ where $f =$ total excess from saturating the negative arcs. $f \leq m$ because each edge's capacity has been incremented by at most 1.

If one scaling step takes $\tilde{O}(m^2)$ time, then total runtime for min-cost flow is $\tilde{O}(m^2 \log U)$ which is weakly polynomial. Can we find a strongly polynomial algorithm? Yes, but it's very complicated.

The algorithm up here is called "capacity scaling algorithm." There's also "cost scaling" algorithm which depends on $\log C$ instead of $\log U$ and we will solve this in our homework problem.

§7.4 Complementary slackness

Back to merchants analogy with feasible prices and reduced costs, when is our network stable? It's when all reduced cost are nonnegative. Why? Suppose reduced cost is positive, we're shipping at a loss, so there should be no flow. Suppose the reduced cost is negative, this is an opportunity for profit and that edge will be saturated. Suppose the cost is 0, it doesn't really matter so this is what is known as "complementary slackness."

Complementary slackness implies optimality, i.e. if we have a graph G such that every positive reduced cost edge is empty and every negative reduced cost edge is saturated, then our flow is optimal.

This suggests a way to find min cost flow. To find feasible prices, we have to saturate all negative cost arcs which would create excesses and deficits then tidy up by sending flow back. Sending flow back shouldn't use positive arcs according to our optimality condition. We don't have negative edges now. So, everything is cost 0 and thus just a max flow problem with no costs.

In a sense, min cost flow = shortest path + max flow. The best running time we've found is about $O(m^2)$ strongly polynomial. To do better than that, we can use double scaling and get $\tilde{O}(mn \log \log U \log C)$.

8 Linear Programming

In min cost flow, we want to minimize $\sum c(e)f(e)$ subject to $0 \leq f(e) \leq u(e)$ and $\sum_{e \text{ into } v} f(e) = \sum_{e \text{ out of } v} f(e)$. We can view $f(e)$ as variables and $u(e)$ as constant and see that both our objective function and the constraints are linear.

A general problem of this kind is find min/max of linear objective subject to given linear constraints. This was formulated in 1940s as a part of war effort. Geore Dantzig developed a "simplex algorithm" to solve these linear programs and established the field of "operations research" and "combinatorial optimization" which encompass both shortest path and max flow problems we've learned in this course.

The theory was behind the practice (the algorithm), however. There were many remaining questions such as: Can it be solved? Can you write a solution and check it? Can you find a solution or show that none exists? Dantzig showed that it's possible to write down the solutions, so linear programming is in NP. The answer can be found by brute force, so it is exponential time in the worst case. In 1970, Khachian came up with ellipsoid algorithm which works in polynomial time but is not practical. A better algorithm is developed in 1985 and is practical.

Linear programming (LP), to David Karger, is the boundary between nice combinatorial problems and complicated general optimizations.

§8.1 Forms of linear programs

Definition 8.1.1 (General form of linear programming)**.** A linear program consists of:

- Variables
- Constraints (linear inequalities in form of $\geq$ or $\leq$).

Vector x is feasible if it satisfies all constraints. LP is feasible if there exists a feasible x. x is optimal if it gives the best objective function value over all feasible points.

Not all LP are feasible. For example, $x > 5$ and $x > 0$. It is possible for LP to be unbounded if there exists a feasible x of arbitrarily good values. For example, maximize x subject to $x > 0$.

Lemma 8.1.2

Every LP is unbounded, infeasible, or has optimal solution.

Proof. This follows from the topological compactness of $\mathbb{R}^n$. We can keep on finding series of improving points. Either there is no limit (so unbounded) or there is a limit that you can converge to and is in the set. $\square$

Definition 8.1.3 (Canonical form)**.** A LP can be written in canonical form as follows: Maximize $c^\top x$ subject to the constraints $Ax \le b$ coordinate-wise. (A is a matrix. $c^\top$, x, and b are vectors.) For each i, $a_i x \le b_i$. c is called the objective.

Claim 8.1.4 — Every LP can be transformed into canonical form.

Proof. Suppose we have $\min d^\top x$, we can change to $\max -d^\top x$. Suppose we have $\sum a_i x_i + b_i \le q_i x + r_i$, rearrange to $\sum (a_i - q_i) x_i \le \sum (r_i - b_i)$. Suppose we have $a_i x \ge b_i$, rearrange to $(-a_i)x \le (-b_i)$. Suppose we have $a_i x = b_i$, write two inequalities: $a_i x \le b_i$ and $a_i x \ge b_i$. $\square$

Definition 8.1.5 (Standard form)**.** A standard form of LP asks us to minimize $c^\top x$ subject to $Ax = b$ and $x \ge 0$ coordinate-wise.

Claim 8.1.6 — Every LP can be transformed into a standard form equivalent.

Proof. We will convert it to canonical form first. So, we have to figure out how to convert canonical form to standard form: change $Ax \le b$ to $Ax = b$ and $x \ge 0$.

Suppose we have $a_i x \le b_i$, add a "slack variable" s_i and write $a_i x + s_i = b_i$ and $s_i \ge 0$.

Let x be positive or negative. Split into $x^+ \ge 0$ and $x^- \ge 0$ where $x = x^+ - x^-$. $\square$

Steps to solve linear programming:

1. Characterize solutions and write down.
2. Checking if there exists a solution.
3. Verifying infeasibility.
4. Find a way to verify optimality.
5. Construct optimal solutions.

§8.2 Solving linear equalities

Suppose we are given a linear system in form of $Ax = b$. When does a solution exist?

For square matrices, we can verify that a solution exist by simply checking a valid solution in $O(n^2)$... sort of. There is a problem: we might not be able to write x down because some values may be irrational.

Lemma 8.2.1
These statements are equivalent:

1. A is invertible
2. $A^\top$ is invertible
3. $det(A) \neq 0$
4. A has linearly independent rows
5. $Ax = b$ has a unique solution for all b
6. $Ax = b$ has a unique solution for some b

§8.2.1 Writing down solutions

An integer n uses $\log n$ bits to represent. A rational number $\frac{p}{q}$ requires $\log p + \log q$ bit to represent. For simplicity, we will call the number of bits required $\text{size}(n)$ and $\text{size}(\frac{p}{q})$ respectively. Notice that $\text{size}(xy) = \text{size}(x) + \text{size}(y)$.

An n-vector has size $n = \sum \text{size}(\text{numbers}) + \log n$. (Need $\log n$ to record the size of the vector itself.) An $m \times n$ matrix requires $\sum \text{entries} \approx mn \cdot \text{size}(\text{an entry})$.

What is the size of a matrix product? $\text{size}(AB) \leq size(A) + size(B)$.

The goal for us solving linear programming problems is to represent the solutions with size polynomial respect to the input size.

Claim 8.2.2 — $det(A)$ has size polynomial to $\text{size}(A)$.

Proof. We have A be an $n \times n$ matrix. We know $det(A) = \sum_{\pi:n\to n} \left(\text{sign}(\pi) \cdot \prod_{i=1}^{n} A_{i,\pi(i)}\right)$ where $\text{sign}(\pi)$ is the parity of the number of swaps. The product is $n \cdot \text{size}(\text{entry})$. So, the entire sum is $n! \cdot (\text{size}(\text{entry})^n)$, which takes only $\log(n!) + n \cdot \text{size}(\text{entry}) = O(n \log n) + n \cdot \text{size}(\text{entry})$ which is polynomial. □

Corollary 8.2.3

$A^{-1} = \frac{1}{det(A)} \cdot \text{cofactor}(A)$. Each entry is in the form $\frac{det(A_{ij})}{det(A)}$ which is polynomial in n. So, the solution to $Ax = b$ is $x = A^{-1}b$ which is polynomial in the input size.

§8.2.2 Finding a solution

We can use Gaussian Elimination or Matrix Inversion which requires a polynomial number of operations, and each operation involves numbers that are small integer multiples of the determinant, meaning the total time is polynomial.

§8.2.3 Generalizing to non-square matrices

When we say $Ax = b$, we are simply asserting that the columns of A span B. This is true if and only if the maximal linearly independent subset of columns span b, then we can extend to a basis by adding more independent columns to get A' which is a square matrix. Thus, we have a unique solution to $A'x = b$, giving us a unique x which zeroes out the added basis columns. We can find and check answer in polynomial time.

Therefore, $Ax = b$ has a simple witness for feasibility, namely one of the valid x solutions.

§8.2.4 Proving there is no solution

We are going back to the idea that $Ax = b$ means columns of A span b. Basically, we need to prove that $b \notin \{Ax \mid x \in \mathbb{R}^n\}$. (This set is a subspace spanned by columns of A.)

In 2D case, the subspace $\{Ax\}$ is a line through origin. To show that b is not spanned by A, we can break b into the components parallel to the line and perpendicular to the line. Let's call the perpendicular portion y where $y \neq 0$. We know $y \cdot b \neq 0$ while $yA = 0$. This is our proof: just give vector y and verify these two conditions.

To construct a proof y: Note that if $y \cdot b \neq 0$ there is some scaled y that has $yb = 1$. Our conditions would be $yb = 1$ and $yA = 0$ which is a simple linear equality system we already know how to solve. This also tells us that size of y is polynomial to A and b.

Now, in the other direction: Suppose $Ax = b$. Then, $y \cdot b = y(Ax) = (yA)x = 0$.

Theorem 8.2.4

$Ax = b$ has no solution if and only if there exists a y such that $yA = 0$ and $y \cdot b \neq 0$.

There is some sort of duality going on here: If you can solve the original system, then you can't solve this dual system. Otherwise, if you can solve this dual system, it shows that the original system is unsolvable.

To summarize, we can both find a solution or prove non-existence and check the proofs in polynomial time.

§8.3 Inequalities and Geometry

Each constraint corresponds to a half space. If we consider a lot of constraints, then we have a polytope defined by intersections of half-spaces. A polytope P is convex when for $x, y \in P$, $\lambda x + (1 - \lambda)y \in P$ (every line segment stays in the polytope).

Claim 8.3.1 — An optimal solution is found in the "corner" of the polytope.

Definition 8.3.2 (Vertex)**.** A vertex is a point that is not a convex combination of any other points in the polytope.

Definition 8.3.3 (Extreme point)**.** An extreme point is a point that gives a unique optimal solution for some objective function c (cost).

§8.3.1 Basic solution

We define a constraint $Ax \leq b$, or $Ax = b$, or $Ax \geq b$ to be tight when $Ax = b$.

Definition 8.3.4 (Basic point)**.** For n-dimensional LP, a point is **basic** if

1. All equality constraints are tight
2. n linearly independent constraints are tight

If a basic solution is feasible, we call it a **basic feasible solution (BFS)**.

We will see that vertex = extreme point = basic feasible solution.

Lemma 8.3.5
Any standard form LP, $Ax = b$, $x \geq 0$ with an optimal solution has one at a basic feasible solution.

Proof. Suppose there exists an optimal solution that is not at a basic feasible solution. This implies there are less than n tight constraints. There is some degree of freedom we can move x while still keeping those tight constraints: We can move in the subspace defined by these tight constraints. Moving can either improve, worsen, or leave the objective unchanged. If we can improve, that means our original assumption of optimality is wrong. If we can worsen, then we can move in the opposite direction which also improves the optimality. Moving can thus only leave the objective unchanged, meaning the entire subspace is optimal.

In standard form, our subspace contains a line through current optimum x, which we will write as $x + \varepsilon d$ for some direction d. In one direction, some coordinate x_i is decreasing ($x \geq 0$ and we are not tight). We can just keep moving in this direction until some x_i hits 0, giving us a new tight constraint. Repeat until we have n tight constraints, giving us a basic feasible solution. □

In canonical form, we might not have an optimum at BFS. For example, maximize x with respect to $x \leq 1$. There is no intersection for us to find BFS.

Corollary 8.3.6

If there exists an x of any given value, then there exists a basic feasible solution that is at least as good as x.

Now, we show that BFS = vertex.

Proof. Suppose a BFS is not a vertex. This means the BFS is a convex combination of two points in polytope. There is a line in polytope through that vertex, all feasible. This means we do not have n tight linearly independent constraints. Then, our point is not a BFS.

Conversely, if we have less than n tight constraints (not a vertex), then their intersection contains a feasible line through the point, then the point is a convex combination of points on line on each side and thus not a BFS. □

Now, extreme point = BFS.

Proof. Suppose we have an extreme point, meaning it is a unique optimum in some direction. This implies that it is a BFS. If not, we do not have n tight constraints, so there is a feasible line through the point we can move without changing the objective, so this extreme point is not actually a unique optimum.

Conversely, if you have a BFS: Let T be the set of tight $x_i \geq 0$ constraints. Define objective $\sum_{x_i \in T} x_i$. Min objective is 0. At other points, objective is > 0. □

§8.3.2 Naive algorithm for solving LP

We know optimum can always be found at BFS. Then, a simple algorithm is to try every BFS. How many BFS are there? Suppose we have m constraints in n dimension. There is at most $\binom{m}{n}$ BFS (picking n constraints to be tight out of all m). We try all BFS: solve the square linear system by Gaussian elimination or computing the inverse (so each optimum has polynomial size). Then, for the points we found, just check feasibility and if they are feasible just compute and keep track of the best objective.

Runtime is $\binom{m}{n} \cdot n^3$ if we use Gaussian Elimination, which is sadly exponential because $\binom{m}{n} \approx m^n$. At least it's finite!

§8.4 Duality: Verifying optimality in polynomial time

This is a decision problem, so we can find the complexity of answering $OPT(LP) \geq k$. We know $LP \in NP$ because as we are presented an optimum, we can check feasibility in polynomial time. How do we prove that $OPT(LP) \not\geq k$, namely $P \in coNP$?

Duality will help provide a succinct prove that we cannot do better than some point, so we have proven that the point is optimal. This is a very powerful result. We have seen some special cases: max-flow min-cut, prices of min cost flow, potentials of shortest paths, Nash equilibrium, etc.

Note 8.4.1. Starting from here, some transposes may be left off. Verify carefully.

Idea: suppose we have $v^* = \min c^\top x$ subject to $Ax = b$ and $x \geq 0$. Let us try to find lower bound on v^*. One techinque we can use is to sum of constraints to get others or multiply by some coefficients, just like in Gaussian Elimination. Try to multiply each $a_i x = b_i$ by some y_i. Adding them up gives $\sum y_i(a_i x) = \sum y_i b_i$, namely $yAx = yb$ for all y.

Find some y such that $y^\top A = c$. Then, $y^\top b = y(Ax) = (yA)x = c^\top x$ for all feasible x. This means all feasible points have the same objective value. So, such y does not exist.

Let us opt for a looser goal: find y such that $y^\top A \leq c$. Then, $yb = y(Ax) = (yA)x \leq c^\top x$ for all feasible x. This means finding such a y gives us a lower bound on the optimal objective: $v^* \geq yb$.

What is the best possible lower bound? Maximize yb with respect to $yA \leq c$. This is called the **dual LP**. The original problem is called the **primal LP**. Let us define $w^* = \max yb$.

We have just proven the **weak duality** problem: $w^* \leq v^*$.

Sanity check: dual of dual LP is primal. Dual is $\max yb$ where $yA \leq c$. Change to minimization and adding slack variables: $\min -by$ with respect to $Ay + Is = c$, which is $\min(-by^+ + by^-)$ with respect to $Ay^+ - Ay^- + Is = c$, $y^+, y^-, s \geq 0$. This can be seen as

$$\min \begin{bmatrix} -b & b & 0 \end{bmatrix} \begin{bmatrix} y^+ \\ y^- \\ s \end{bmatrix}$$

subject to

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

Dual is $\min cz$ such that

$$\begin{bmatrix} A & -A & I \end{bmatrix} z \leq \begin{bmatrix} -b & b & 0 \end{bmatrix}$$

This simplifies to $Az = b$, $z \leq 0$. If $x = -z$. Then, we have $\min cx$, $Ax = b$, $x \geq 0$. So, dual of the dual is the primal.

Corollary 8.4.2

We have a primal LP P and its dual D. If P is feasible, then D has an upper bound, meaning D is either infeasible or bounded. If P is feasible and unbounded, then the dual is infeasible. Then, we can go the other way around. So, we have four cases:

1. Both bounded and feasible
2. Both infeasible
3. One infeasible, one unbounded
4. The other way around

Shorthand: If P is unbounded, we write $v^* = -\infty$, implying D is infeasible. If P is infeasible, we write $v^* = +\infty$., implying P is unbounded.

§8.5 Strong Duality

Karger intuitively proved this with physics and I have absolutely no idea what is going on so I will just jump right into the formal proof.

Consider optimal y^* for dual $\min\{yb \mid yA \geq c\}$. Let us choose a maximal linearly independent subset S of the tight constraints (columns of A). $|S| \leq m$ where m is dimension of y. Let A_s and c_s be the restriction of A and C to S. We still have $y^*b = \min\{yb | yA_s \geq c_s\}$ unchanged. This means we can just forget the original A and consider A_s and assume that all columns are linearly independent.

In other words: Suppose we are able to prove strong duality with respect to A_s, then we have proven that for some x_s^*, we have $A_s x^* = b$, $x_s^* \geq 0$, $c_s x_s^* = yb$. We can recover full A. Let $x^* = x_s^*$ with 0 for all other coordinates. Then, $cx^* = c_s x^*$, $Ax^* = A_s x_s^* = b$, $x^* \geq 0$.

Claim 8.5.1 — There exists an x^* such that $Ax^* = b$.

Proof. Suppose there is no such x^* for the sake of contradiction. The "duality" for linear equalities (as in previous lecture) says that there exists some z such that $zA = 0$ but $zb \neq 0$. Without loss of generality, let $zb < 0$.

Consider $y' = y^* + z$. Then, $y'A = (y^* + z)A = y^*A = c$ which is feasible. The value is $y'b = (y^* + z)b = y^*b + zb < y^*b$ which means y' is a better solution. □

Claim 8.5.2 — $yb = cx$

Proof. We know $Ax^* = b$ and $y^*A = c$, so $yb = y(A^*x) = (yA^*)x = cx$. □

Claim 8.5.3 — $x \geq 0$

Proof. Suppose some $x_i < 0$. Let $c' = c + e_i$. Consider a solution to $y'A = c'$. Note that $c' \geq c$, implying $y'A \geq c$, so y' is feasible for original LP. Value is $y'b = y'(Ax^*) = (y'A)x^* = c'x^* = cx^* + e_i x^* < cx^*$ because $x_i < 0$. y^* is not optimal (pushing the wall up lowers the ball), so this is a contradiction. □

Intuitively, x^* tells us how much optimal solution will change if we modify position c_i of corresponding constraints. Increasing c_i corresponds to tightening the constraint.

Corollary 8.5.4

Optimization = Finding a feasible point.

Given a LP $\min cx$ wrt. $Ax = b$, $x \geq 0$ and its dual $\max yb$ wrt. $yA \leq A$. If we put the duals together, then we are basically solving the system $Ax = b$, $x \geq 0$, $yA \leq c$, $cx = by$. This means any feasible solution here is feasible for both LPs and achieves the matching objective, so it is optimal for both LPs.

§8.6 Rules for Duals

For standard primal LP, we can find a canonical dual LP. Can we dualize **any** LP?

Primal LP: Let $v^* = \min(c_1x_1 + c_2x_2 + c_3x_3)$ where $x_1 \geq 0$, $x_2 \leq 0$, and x_3 is UIS (unrestricted in sign), and for the constraints:

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

The corresponding dual: $w^* = \max(y_1b_1 + y_2b_2 + y_3b_3)$ subject to

$$\begin{aligned} y_1A_{11} + y_2A_{21} + y_3A_{31} &\leq c_1 \\ y_2A_{12} + y_2A_{22} + y_3A_{32} &\geq c_2 \\ y_1A_{13} + y_2A_{23} + y_3A_{33} &= C_3 \end{aligned}$$

where y_1 UIS, $y_2 \geq 0$, $y_3 \leq 0$.

Table 8.1: Simple conversion between primal minimization LP and dual maximization LP

	Primal min	Dual max	
constraint	$\geq b_i$	≥ 0	variable
	$\leq b_i$	≤ 0	
	$= b_i$	UIS	
variable	≥ 0	$\leq c_j$	constraint
	≤ 0	$\geq c_j$	
	UIS	$= c_j$	

Intuitively, adding a primal constraint makes it harder for primal LP to be harder, so the minimum objective should be higher. This is reflected in the dual, as adding a variable allows for the dual to have higher value as it tries to maximize the lower bound for the primal minimization.

In another sense, tighter primal constraints lead to looser dual variables. Inequalities lead to sign restrictions. Equality constraints lead to no restrictions.

The natural constraint constraints to non-negative variables. In physics, lower bound on min means non-negative forces. Upper bound will lead to non-positive forces.

§8.7 Applications of duality

§8.7.1 Shortest path

Think of "lifting the graph up from the table" analogy. We can see that the shortest path is like trying to maximize the distance between t and the lifted s.

$w^* = \max(d_t - d_s)$ subject to $d_j - d_i \leq c_{ij}$ for all i, j (edges).

This is our dual. Let us change to primal: each edge constraint will become variable, and each vertex distance variable becomes constraint.

In the primal, objective is $\sum c_{ij} y_{ij}$. The constraints are $\sum_j y_{ji} - \sum_j y_{ij} = -1$ if $i = s$, $+1$ if $i = t$, and 0 otherwise. $y_{ij} \geq 0$.

This is a flow problem! Non-negative flow, conserved at every vertex except one flow at s and t. The objective minimizes the cost under c_{ij}.

§8.7.2 Maximum flow

For simplicity, we are turning this problem into circulation by adding (t, s) edge and solving for circulation maximizing the flow on (t, s). This makes our lives a bit easier because we do not have to write special cases for the conservation conditions.

Let x_{vw} be the flow on edge (v, w). Our LP is:

$$\begin{aligned} w^* &= \max x_{ts} & \\ \sum_w x_{vw} - x_{wv} &= 0 & \text{(balance)} \\ x_{vw} &\leq u_{vw} \forall e = (v, w) & \text{(capacity)} \\ x_{vw} &\geq 0 & \text{(lower bound)} \end{aligned}$$

Then, its dual would have a variable per edge capacity constraint y_{vw}, and a variable per balance constraint z_v. To understand how the variables would turn into constraints, we write out the matrix of coefficients of the constraints. Notice that the rows can be separated into edge capacity rows and vertex balance rows. In each edge capacity row, only a single 1 appears to represent x_{vw} when multiplied with the variable vector. In each balance constraint, multiple $+1$ appear for outgoing edge to w and -1 for incoming edge from u entering v. Looking at the columns, we can see the number of columns correspond to the number of edges (variables $x_v w$). Each column has 1 for y_{vw} variables, and for z_v variables, we have $+1 z_v - 1 z_w$ for each edge (v, w). The right hand side of these new constraints are going to be 0 for $(v, w) \neq (t, s)$ and 1 for $(v, w) = (t, s)$. The variables are natural, $x_{vw} \geq 0$, so the relationship in the constraints is $\geq$ because we are dealing with minimization problem. Now, for the dual variables, $y_{vw} \geq 0$ and z_v are unrestricted in sign.

The dual LP is:

$$\begin{aligned} v^* &= \min \sum u_{vw} y_{vw} \\ z_v - z_w + y_{vw} &\geq 0 && \text{for } (v,w) \neq (t,s) \\ z_t - z_s + y_{ts} &\geq 1 \\ y &\geq 0 \\ z &\in \mathbb{R} \end{aligned}$$

How can we interpret this? Notice that the constraints look like triangle inequality. We can see y_{vw} as lengths and z_v as distances. Without loss of generality, assume $y_{ts} = 0$ (otherwise we can set $u_{ts} = \infty$ when doing max flow but we want to do minimization). So, the second constraint is $z_t - z_s \geq 1$, meaning the distance between s and t needs to be at least 1. Without loss of generality, let $z_s = 0$ (shortest path start). Thus, the problem is like trying to place nodes along a number line from 0 to 1 but the edges (y_{vw}) need to have greater length than the distance (triangle inequality, $y_{vw} \geq z_w - z_v$). This is called an **embedding problem**. If we look at each edge as a pipe, u_{vw} is the cross section area and y_{vw} is the length, thus we are minimizing the total volume of the network.

Suppose we send f units of flow from s to t. Each unit of flow is 1 unit long and has 1 unit area, so the volume is 1. Therefore, the minimum volume of this network embedding is equal to the volume of the flow in the network. To find the optimal edge lengths, we introduce **complementary slackness**.

§8.8 Complementary slackness

Given feasible x and y for primal and dual LPs. Define $cx - yb$ as the duality gap. Weak duality tells us that the gap is ≥ 0. Strong duality tells us that the gap is 0 at optimum.

Suppose the dual is $\max \{yb \mid yA \leq c\}$. We can rewrite this into $\max \{yb \mid yA + s = c, s \geq 0\}$ with s as the slack variables. Then, $cx - yb = (yA + s)x - yb$. The primal is $\min \{cx \mid Ax = b, x \geq 0\}$. So, we can substitute $Ax = b$ and have the gap be $(yA + s)x - y(Ax) = sx$. Since $s \geq 0$ and $x \geq 0$, then $sx \geq 0$. If $sx = 0$, meaning $\sum s_i x_i = 0$, then either $s_i = 0$ or $x_i = 0$. This means that either the variable is 0 ($x_i = 0$) or the corresponding constraint is tight ($yA_i = c_i$).

§8.8.1 Maximum flow (repeated)

Consider optimal variable values: y^* and z^*. We will have $y_{vw} = z_w - z_v$ because we want to minimize $\sum u_{vw} y_{vw}$ but we are restricted by the lower bound triangle inequality condition.

Define cut $S = \{v \mid z_v^* < 1\}$. We leverage complementary slackness. Suppose (v, w) leaves (goes out of) the cut S. Then, $y_{vw} \geq z_w - z_v > 0$ because $z_v < 1$ and $z_w \geq 0$ so the corresponding capacity constraint has to be tight: $x_{vw} = u_{vw}$. Suppose (v, w) enters the cut S. $z_v > z_w$ and $y_{vw} \geq 0$ imply $z_v + y_{vw} > z_w$, meaning the constraint is slack. Therefore, the variable $x_{vw} = 0$.

We learn that every edge is saturated and every entering edge is 0. By flow conservation, the value of the cut is the value of the flow. (Flow cannot come back because every entering edge is 0.)

Any cut gives a dual solution: just set $y_{vw} = 1$ for each cut edge.

Note that we cannot apply LP strong duality directly because the optimal solution to the minimization LP might not be a valid cut. Leveraging complementary slackness allows us to transform any fractional LP solution into a cut.

§8.8.2 Min cost circulation

Change the primal objective to $\min \sum c_{vw} x_{vw}$. The constraints are still the same. The dual objective turns into a maximization problem. Therefore, the constraints' right hand side would change into $\leq c_{vw}$, and we will have $y \leq 0$.

Rewrite $p_v = -z_v$. Then, we have dual LP: $\max \left\{\sum u_{vw} y_{vw} \mid y_{vw} \leq c_{vw} + p_v - p_w, y \leq 0\right\}$. Think of reduced costs! The objective is just the sum of negative reduced costs.

If $x_{vw} < u_w$, then the constraint is tight, namely $y_{vw} = 0$. If the reduced cost is negative, $y_{vw} < 0$ is slack, then $x_{vw} = u_{vw}$. If the reduced cost is positive, then $y_{vw} < c_{vw} + p_v - p(w)$ (the constraint is slack), then $x_{vw} = 0$.

Just by applying duality, we get feasible price functions and complementary slackness characteristics we derived directly earlier in the class!

9 Algorithms for linear programming

§9.1 Simplex algorithm (1940s)

§9.1.1 Notation juggling

We already have an algorithm. Just try every basic feasible solution (BFS). There are at most $\binom{m}{n}$ BFS, so the runtime is exponential.

Simplex algorithm tries to be smarter about searching. We look at the standard form $\min\{cx \mid Ax = b, x \geq 0\}$ and assume without loss of generality that A has full row rank.

Recall that at a BFS, all m equality constraints are tight, and at least $n - m$ inequality constraints are tight. Also, these constraints form basis with a_i in n dimensions.

If we write out the matrices, we can see that m rows correspond to the equality constraints. Then, other n rows form the identity matrix for $x_i \geq 0$ constraints, where $n - m$ of them are tight and m might not be. Then, we can separate the matrix into top m rows and bottom m rows, and left $n - m$ columns and right m columns.

At BFS, we need linearly independent top n rows. Notice that the first $n-m$ columns are clearly independent because of the identity matrix part below. But, the last m columns have all zeroes, so their independence rely completely on the full matrix.

Definition 9.1.1 (Basic feasible solution (Alternate))**.** x is a basic feasible solution if and only if $Ax = b$ and m linearly independent columns of A constraints correspond to each slack $x_j > 0$.

Definition 9.1.2 (Basis)**.** The set of last m columns is called a basis. The corresponding x_j variables are basic (B), and the others are non-basic (N).

Given B, we can compute x: Let us write $A_B =$ the B columns of matrix A. We have A_B of size $m \times m$ with full rank. We can solve $A_B x_b = b$. Set $x_N = 0$.

Note that we can have multiple bases B for the same vertex x, because we might have more than $n - m$ tight inequality constraints.

Thus, x is a vertex if there exists a basis B such that $x_N = 0$, A_B is $m \times m$ non-singular matrix, and $x_B = A_B^{-1} b \geq 0$ feasible.

§9.1.2 The algorithm

We start at a basic feasible solution then try to move to a better one by following an edge until we cannot. An edge is defined by $n-1$ tight constraints.

Rewrite $Ax = b$ as

$$\begin{pmatrix} A_N & A_B \end{pmatrix} \begin{pmatrix} X_N \\ X_B \end{pmatrix} = b$$
$$A_N x_N + A_B x_B = b$$

This is true for any feasible x, not just vertex. Rearranging, we have:

$$x_B = A_B^{-1}\left(b - A_N x_N\right)$$

Objective is

$$\begin{aligned} cx &= c_B x_B + c_N x_N \\ &= c_B A_B^{-1}\left(b - A_N x_N\right) + c_N x_N \\ &= c_B A_B^{-1} b + \left(c_N - c_B a_B^{-1} A_N\right) x_N \end{aligned}$$

Notice that the left term is a constant, and the right term only depends on x_N, not the basis at all. Objective can only improve or worsen based on the second term.

Definition 9.1.3 (Reduced cost)**.** Given a basis B, the reduced cost is:

$$\tilde{c_N} = c_N - c_B a_B^{-1} A_N$$

This results in $cx = \text{some constant} + \tilde{c_N} x_N$ Notice that if $\tilde{c_N} \geq 0$, then no feasible points have smaller objective, because our vertex has all $x_N = 0$. So, we are at optimum if all reduced costs are non-negative.

Suppose there is some non-basic $c_j < 0$. Then, we can continuously improve our solution by increasing x_j to contribute negatively to the objective. Note that this modifies x_N and thus also x_B because $x_B = A_B^{-1}\left(b - A_N x_N\right)$. Thus, we can keep going until we hit $x_i = 0$ for some basic $x_i \in x_B$. x_j is now slack but x_i becomes tight, so the basis is changed! This vertex is called a "pivot."

Simplex algorithm is thus simple: just pivot until you find an optimum. We need to do linear algebra over $m \times m$ matrix and test n constraints to find whichever one is tight first. Runtime is polynomial in n and m.

There is a potential problem of not being able to move. Maybe increasing non-basic x_j hits some basic x_i that is already 0. Okay, we added x_j to the basis and removed x_i, but the objective function did not change at all. We might be stuck in a cycle of pivots that never improve our objective. We have to design a guaranteed non-cycling pivot rule. Is there one? There is, called the "lexicographiclly first" rule. We never visit some basis more than once, so runtime is dependent on the number of pivots: $\binom{m}{n}$.

We also have to find a starting vertex. We can just find a feasible point and round it to get a vertex but this is as hard as optimizing... in general! For some specific polytopes, the problem can be easier. We can find the feasible point by optimizing a different LP. We will see this in the homework.

§9.1.3 Is this polynomial?

It is not apparently polynomial. Theoreticians have been working on this for quite a while.

Klee-Minty came up with a hypercube twisted so that all vertices are out of line so their objective values differ. They showed that many pivot rules visit all the vertices of this cube. Running time is thus exponential in the lower bound for these rules.

There might be a polynomial pivot rule, though.

Conjecture 9.1.4 (Hirsch conjecture)**.** The max diameter of a polytope in simplex steps is only $m + n$.

Unfortunately, this was proven by Santos that the diameter is at least $(1 + \varepsilon)(m + n)$ in 2010.

Kaloi-Kleitmen showed that the path length is at most $m^{\log n}$ which is a tremendous improvement from m^n. Then, simplex runtime is $2^{\sqrt{n}}$.

Spielman-Tengi used smoothed analysis to show that any linear programming with small random perturbations has fast simplex algorithm.

§9.1.4 Simplex and Duality

Recall, reduced cost is $\tilde{c_N} = c_N - c_B A_B^{-1} A_N$. When all $\tilde{c_N} \geq 0$, we have an optimum.

Define $y = c_B A_B^{-1}$, so $c_B = yA_B$. Then, $yA_N = c_B A_B^{-1} A_N = c_N - \tilde{c_N}$, which is $\leq c_N$ at optimum. Then,

$$\begin{aligned} y \begin{pmatrix} A_N & A_B \end{pmatrix} &\leq \begin{pmatrix} c_N c_B \end{pmatrix} \\ yA &\leq c \end{aligned}$$

Also, $yb = c_B A_B^{-1} b = c_B x_B = cx$ since $c_N x_N = 0$ because $x_N = 0$.

y is a dual optimum! To compute y, we only need the basis. The simplex algorithm which finds the basis for optimum x also finds the optimum y at the same time! More generally, any basis gives some y. This idea gives us another proof of strong duality, which proves that the simplex terminates.

§9.2 Ellipsoid Method

As good as simplex algorithm is, we do not know any polynomial theoretical time bounds except for special cases such as network simplex and in smoothed analysis. Ellipsoid method was thus developed by Khachiyan to achieve a good algorithm in a theoretical sense.

The ellipsoid algorithm is based on the binary search algorithm for finding an integer between secret upper/lower bounds. (We aren't given the full bounds but only the starting lower or upper bound.) The search can go on forever, so we need an algorithm that has error tolerance in form of the number of bits in the answer and generalizes to higher dimensions.

Ellipsoid algorithm is a feasible point algorithm, not an optimization algorithm. This is fine because we have learned that we can reduce optimization problem to feasible point problem.

§9.2.1 The algorithm

Suppose there is a polytope (we do not know where it is). How can we find a point inside the polytope? We are given a starting bound: an ellipsoid with contains the feasible polytope. One way is to pick a hyperplane and ask which side, then draw a smaller ellipsoid containing the polytope. Keep shrinking the ellipsoid until it hits the polytope and reveals a feasible point.

Specifically, we claim that when we have an ellipsoid containing the polytope, the point in the center is in the polytope otherwise there exists a separating hyperplane through the center and the polytope lies entirely on one side. We can draw an ellipsoid on that side. Finding the separating hyperplane can be done through finding the violated constraint and translating it to center.

Definition 9.2.1 (Ellipsoid)**.**

$$E(D, z) = \left\{x \mid (x-z)^\top D^{-1}(x-z) \leq 1\right\}$$

where $D = BB^\top$ for some invertible B. This can be viewed as transformation from coordinate x to $Bx + z$ (from unit sphere to ellipsoid). This is a generalization of $(x - x_0)^2 + (y - y_0)^2 \leq 1$.

Assume w.l.o.g. that we start with a unit sphere and vertical hyperplane. We can create a smaller ellipsoid on the correct side. We know that the center should be at $(\varepsilon, 0, 0, 0, \ldots)$ for some number ε (shifted from the original center).

$$d_1^{-1}(x_1 - \varepsilon)^2 + \sum_{i>1} d_i^{-1} x_i^2 \leq 1$$

Note that the volume is $\sqrt{\prod d_i}$. (Square root because in the formula we operate based on squared x_i coordinates.)

Let us check that other points in the half-sphere are contained in our new ellipsoid. For $(1, 0, 0, \ldots)$: $d_1^{-1}(1-\varepsilon)^2 \leq 1$ so $d_1 = (1-\varepsilon)^2$. For $(0, 1, 0, \ldots)$: $\frac{\varepsilon^2}{(1-\varepsilon)^2} + d_2^{-1} = 1$ by Pythagorean theorem, so $d_2^{-1} = \frac{\varepsilon^2}{(1-\varepsilon)^2} \approx 1 - \varepsilon^2$. Same for $d_3, \ldots$. Thus, the volume is about $\frac{1-\varepsilon}{(1-\varepsilon^2)^{\frac{n}{2}}}$.

Set $\varepsilon \approx \frac{1}{n}$. We will get volume of about $\frac{1-\frac{1}{n}}{1-\frac{1}{2n}} \approx 1 - \frac{1}{2n}$.

§9.2.2 Runtime analysis

Given constraints, can we make an ellipse containing the feasible region? Yes! We proved that every vertex has a polynomial size in n. If we can bound the number of bits to d, then we only need a sphere of radius 2^d.

Let us consider the ending size of the ellipsoid. Suppose it has full dimensions (not degenerate), then there are $n + 1$ affinely independent vertices, all with polynomial size coordinates. The volume is the determinant of these coordinates which is also polynomial size, so at least $2^{-\text{poly}(n)}$.

Starting with $2^{\text{poly(n)}}$ size and shrinking to $2^{-\text{poly}(n)}$ with ratio of $1 - \frac{1}{n}$ per step or alternatively, $\frac{1}{e}$ per n steps, takes polynomial time.

If the ellipsoid does not have full dimensions, we can use error bound. For example, if the linear program is in form of $Ax = b, x \geq 0$, we write it as $|Ax - b| \leq \varepsilon$ with sufficiently small ε to ensure we do not query other lattice points not on $Ax = b$.

§9.2.3 Grotschel Louasz Schriver

We already learned that feasibility can be used to solve optimization problems. Ellipsoid algorithm is even more interesting, as we have seen: finding separating hyperplane leads to finding feasible points and thus optimization. (Given any infeasible point, find a violated constraint.) Basically, we want to create a separation oracle for LP problems with exponentially many constraints. Ellipsoid is bizarre because it can find the feasible point for any exponential constraints after looking at only polynomial number of points. Note that you can also use optimization to solve separation problem.

§9.3 Interior point algorithm

Ellipsoid algorithm runs in about $O(n^6)$ which is somewhat slow. In 1985, Karmaker came up with interior point algorithm which makes use of a "potential function" analogous to pushing magnets (not amortized analysis).

Suppose the LP is in the form of $\min\{cx \mid Ax = b, x \geq 0\}$. We want to minimize $G(x)$.

Definition 9.3.1 (Logarithmic barrier function)**.**

$$G(x) = cx - \mu \sum \ln x_i$$

For tiny x_i, $-\ln x_i$ is huge and makes $G(x)$ close to $+\infty$.

Start with large positive x, we can head toward minimum $G(x)$ by using gradient descent while projecting onto $Ax = b$ subspace. Note that μ is decreased as time goes on.

10 Approximate algorithms

Until now, we have been looking at tractable problems which we focus on "how fast" we can get. Now, we will look at intractable problems where we are more concerned with "how good" our solutions can get because doing things quickly is impossible or impractical.

These are some viable alternatives to **exact solutions**:

- Special cases
- Random inputs
- Slightly non-polynomial bounds ($2^n \to \left(\frac{3}{2}\right)^n$)
- Heuristics (fast in practice)

Here, we will study the approximation algorithms.

We have an optimization problem which is defined by its instances I, solutions $S(I)$ to the instances, and the value function $f : S(I) \to \mathbb{R}$ we want to maximize, giving us $\mathrm{OPT}(I)$.

For example, we have a bin packing where we are given items of variable sizes to distribute into bins of size 1 and we want to minimize the number of bins.

Some technical assumptions: we assume that our inputs and the range of f are integers or rational numbers. $f(\sigma)$ is a polynomial size number (else we cannot write down the solutions). Runtime is polynomial in the bit sizes of numbers.

NP-Hardness is more about decision problems that optimization problems. However, any optimization problems can be turned into decision problems via binary search. So, we call an optimization problem NP-hard if its corresponding decision problem is NP-complete.

§10.1 Quality of an approximation algorithm

We find a way to define the quality of a specific instance, then the quality of the whole algorithm is defined by the quality of the worst case.

Definition 10.1.1 (Absolute approximation)**.** A is a k-absolute approximation algorithm if for each instance I, $|A(I) - \mathrm{OPT}(I)| \le k$.

For example, the vertex coloring problem in the case of planar graphs. If we follow the proof of the four-color theorem algorithmically, we have a 3-absolute-approximation algorithm because in the worst case the graph requries only one color but our construction uses four colors. Actually, we can say 1-absolute-approximation because a graph can only be 1-colorable when there is no edge and 2 and we can change our algorithms to consider those cases.

There is also a problem of edge coloring. Vizing proved that every graph of degree Δ can be colored Δ or $\Delta + 1$, so we can solve this with 1-absolute-approximation algorithm.

Absolute approximation is often impossible. For example, with knapsack problems, suppose we have a k-absolute approximation algorithm. We will prove that this can be turned into an exact algorithm, implying $P = NP$. First, multiply all profits by $k + 1$. Find k-absolute approximation. This gives us the optimal solution because the objective is a multiple of $k + 1$.

Another example if the maximum independent set problem, which doesn't even involve numbers. Suppose we have a k-absolute approximation algorithm. We can just make $k + 1$ copies of the graph. By pigeonhole principle, the approximate solution will have at least one group with $\geq$ OPT selected vertices.

Definition 10.1.2 (Relative approximation)**.** An α-approximation solution for I has value between $\frac{\text{OPT}(I)}{\alpha}$ and $\text{OPT(I)} \cdot a$. (At most $\alpha \cdot \text{OPT}(I)$ for minimization and at least $\frac{\text{OPT}(I)}{\alpha}$. For most cases, $\alpha > 1$. (Some scientists invert and use $\alpha < 1$.)

How would you prove approximate ratio? You cannot compare to the optimum because you do not know the optimum. The trick is to compare to the lower or upper bound on the optimal solution instead.

§10.2 Greedy Algorithms

Consider the max cut problem: We want to find the cut with the most crossing edges.

One solution is to be greedy. Look at the graph one vertex at a time. Place each vertex on the side which is opposite to the most of previously placed neighbors.

The bound on optimal solution is obviously m. On our greedy solution, when we place a vertex, some edges are "decided" as cut or not cut and cannot change later. This means at least half of the edges enter the cut. Via aggregate, we have at least $\frac{m}{2}$ cuts at least. So, our greedy algorithm is 2-approximation.

Consider another problem of set cover: Given a set of n items and m subsets of these items, find a collection of these subsets that cover all the items.

Our greedy algorithm is to iteratively choose the subset that has the most uncovered items.

Suppose that the optimal solution uses k sets. Then, our step covers at least $\frac{n}{k}$ items. At most $\left(1-\frac{1}{k}\right)n$ items remain. After t steps, $\left(1-\frac{1}{k}\right)^t n$ items remain. If this amount is less than 1, then the number of uncovered items is 0 because the number is integer. Approximately, we have $t > \frac{\ln n}{-\ln\left(1-\frac{1}{k}\right)} \geq k \ln n$ because $\ln\left(1-\frac{1}{k}\right) \approx \frac{1}{k}$. So, we have $O(\log n)$ approximation algorithm.

Consider the vertex cover problem: pick a subset of vertices to cover all edges. This is basically a set cover problem where each vertex is a set of its incident edges. So, this can be solved with $O(\log n)$-approximation algorithm.

There is actually a better analysis for vertex cover problem. Given an edge (v, w), we know v or w is in the optimum. We can put both vertices into the cover. We can charge these two vertices against the optimum's one, so we have a 2-approximation algorithm.

Notice that the complement of the vertex cover is an independent set. Interestingly, however, there is no relative approximation for maximum independent set problem. Generally, complementary problems do not have good approximations because if we invert the problem and it becomes emptier, then the approximation ratio is bigger.

§10.3 Scheduling theory

We are given a collection of n jobs to run on m machines ($m = 1$, identical machines, related machines, or unrelated machines with processing times). We want to optimize something (e.g. total runtime, average completion time, the waiting time) with respect to a variety of constraints (e.g. preemption, release times, deadlines).

Here we consider a simple problem where we have parallel identical machines (P) and we want to minimize the last completion time, denoted by $P \mid\mid C_max$.

Let us assign one job at a time. We should assign to the least loaded machine (Graham's Rule). Note that the average load of a machine is $\frac{\sum p_i}{m} \leq \text{OPT}$ but this is not a good time bound when $n = 1$ but m is large (optimum is much greater than average time). Another bound we can use is the longest job but this is also bad if we have a lot of tiny jobs with low number of machines.

Notice that these lower bounds get bad in opposite cases. We can try considering the max of both of them, or more simply, the sum. Let us denote the average load by L and the maximum job p_j. We know $\text{OPT} \geq p_j$ from previous bound.

Consider the max load machine when we are adding the last job. By definition, other machines have at most this machine's load. So, this machine's load is currently $\leq L \leq$ OPT (else it won't be minimum). When we add the maximum job p_j, the load increases by $\leq$ OPT. So, the final load is $\leq 2 \cdot \text{OPT}$. So, we have 2-approximation.

Refine: when we add p_i to machine with load L', the average load becomes at least $L' + \frac{p_j}{m} \leq \text{OPT}$. Notice that we can rearrange the desired sum: $L' + p_j = (L' + \frac{p_j}{m}) + p_j(1 - \frac{1}{m}) \leq (2 - \frac{1}{m}) \cdot \text{OPT}$. So, we have $\left(2 - \frac{1}{m}\right)$-approximation algorithm.

One important observation about this algorithm is that it is online. Furthermore, the bound is tight: we can construct the worst case by scheduling a bunch of tiny jobs and then a huge job. Our algorithm performs badly because it did not leave a machine blank for the incoming large job.

Another idea we can use is called the **Longest processing time (LPT)** strategy which is $\frac{4}{3}$-approximation.

§10.4 Approximation schemes

There is a notion of approximation hardness we can use to understand the difficulty of a problem. Some problems have a known lower bound ≥ 1, so we can use reductions to prove difficulty, just like proving NP-completeness.

An **approximation scheme** for a problem is a family of algorithms A_ε such that A_ε is a $(1+\varepsilon)$-approximate algorithm running in polynomial time. We need to be careful about this. For example, if our algorithm has a runtime of $n^{\frac{1}{\varepsilon^2}}$, it is polynomial time for each ε but not polynomial in ε itself as $\varepsilon \to 0$. Thus, we usually talk about **fully polynomial approximation scheme (FPAS)** where the runtime is polynomial to both n and ε.

FPAS actually ties with pseudo-polynomial algorithms. Consider the knapsack problem with integer profits and max bag size (weight) S. One way we can solve this is with dynamic programming where $B(j,p) =$ the smallest size set from $1, \ldots, j$ that achieves profit p. Then, $B(n,p) =$ the smallest size subset that achieves profit p. If we can compute for every p, we can just binary search for p such that $B(n,p) \leq S$. The recursive formulation is $B(j+1,p) = \min\{B(j,p), B(j, p-p_{j+1}) + s_{j+1}\}$. We can solve this iteratively in $O(mU)$ time where $U =$ maximum profit which is pseudo-polynomial. Thus, knapsack is **weakly NP-hard**.

There are other **strongly NP-hard** problems where there is not even a polynomial time algorithm if the numbers are small. Vertex cover is an example because it does not even have numbers. Even then, there are also other problems which involve values but are strongly NP-hard anyway.

For knapsack problem, one way to approximate the solution is to scale all the large profits down to $\lfloor \frac{n}{\varepsilon P} p_j \rfloor$. So, in the old optimal solution, the new profit is at least $\sum \left(\frac{n}{\varepsilon P} p_j - 1\right) = \frac{n}{\varepsilon} - n$. The new optimal solution is at least this value. The largest $p_j \leq \frac{n}{\varepsilon}$ which is polynomial, so we can use a pseudo-polynomial algorithm to solve it in runtime $O\left(\frac{n^2}{\varepsilon}\right)$. Back to the original profits, we have $\geq (1-\varepsilon)P$. So, this is $(1-\varepsilon)$-approximation in $O\left(\frac{n^2}{\varepsilon}\right)$.

Wait, how do we know P? We do not know the optimal solution after all. In actuality, we only need the lower bound on P which is $\max p_i$. Doing a similar analysis, we will get runtime $O\left(\frac{n^3}{\varepsilon}\right)$.

There is a faster solution. We guess P and try to run the algorithm. If the guess $P \geq \frac{\text{OPT}}{1-\varepsilon}$. The optimal solution scales to value $< \frac{n}{\varepsilon} - n$. So, the pseudopolynomial will not find a solution of value $\frac{n}{\varepsilon} - n$. This will tell us that the guess is too big. If we guess too small, there might be better solutions. So, binary search starting in $[\max_{s_i<S} p_i, \sum p_i]$. We can even get the number of phases down to $\log\log n + \log \frac{1}{\varepsilon}$. (See class notes.)

Pseudo-polynomial algorithms lead to FPAS. FPAS lead to pseudo-polynomial algorithms. Suppose input uses integers bounded by t and have n items, so the optimal value is $\leq nt$. We can set $\varepsilon = \frac{1}{1+nt}$, and our approximate solution would be the optimal solution because of the gap lower bound.

§10.5 Dealing with strongly NP-hard problems

We can use **enumeration** to solve these kind of problems. Enumeration is basically a controlled brute force.

Back to $P \mid\mid C_max$ problem. We schedule the largest k jobs optimally by brute force (because large jobs tend to be problematic), then use Graham's rule for the remainder. Claim: Graham part adds one job to a machine with at most average load $\leq$ OPT. When we add that job, the machine has time $\leq \text{OPT} + p_j$.

Consider the machine with the worst completion time c_j. If c_j is caused by one of the k largest jobs, we are optimal. If it is caused by a smaller job, $C_{\max} \leq \text{OPT} + p_{k+1}$ where p_{k+1} is the added job. Note, $\text{OPT} \geq \frac{kp_{k+1}}{m}$. Thus, $P_{k+1} \leq \frac{m}{k}\text{OPT}$. So, our solution is $\leq \left(1+\frac{m}{k}\right) \cdot \text{OPT}$ with runtime $O\left(m^k \cdot n\right)$. If m is a fixed constant, this is polynomial time for any k. We can get arbitrarily close to the optimum with our selection of k so we have a PAS (not a FPAS).

§10.6 Smarter enumeration technique

For $P \mid\mid C_{\max}$ problem, instead of trying to minimize the runtime we can consider the decision version: Is it possible to complete all tasks in time T (construct a solution)? Then, we can optimize this by binary searching over T. The lower bound is $\max\{p_max, \frac{\sum p_j}{m}\}$, and the upper bound is $\sum p_j$. The ratio of these bounds is m. So, we can test out $\text{LB}, \text{LB}(1+\varepsilon), \text{LB}(1+\varepsilon)^2, \ldots, \text{UB}$. There are $\log_{1+\varepsilon} \frac{\text{UB}}{\text{LB}} = \frac{\log m}{\log(1+\varepsilon)} = \log_\varepsilon m$ such values. So, binary search takes $\log\log_\varepsilon m$ steps.

Approximate decision problem suffices. Given a solution of time T, we can give a solution of time $\leq (1+\varepsilon)T$. So, let us find an algorithm that solves this decision problem. We will combine the enumeration with rounding technique.

The challenge with enumeration is that your bruteforce solution space may be large. So, we consider the special case where there are k distinct job sizes. Define a "machine type" as a vector of the number of jobs of each size where the total size is $\leq T$ (instead of listing the jobs so the sum of jobs is $\leq T$). There are at most n^k types to describe a machine. The number of problem instances for a given number of machines m is also no more than n^k.

Note the substructure: scheduling on m machines is equivalent to scheduling on $m-1$ machines plus 1 machine. This suggests a dynamic programming approach. For $r := 1$ to m, we enumerate all inputs that are feasible in time T for r machines. For $r = 1$, the answer is simply all machine types. If we know the answer for r, we can generate the answers for $r+1$ by taking the cartesian product of r-feasible inputs and the number of machine types. The size is at most $n^k \cdot n^k$ so there are at most n^{2k} pairings. Total runtime is $O\left(mn^{2k}\right)$ which is polynomial for any constant k.

What do we do when face with many job sizes? Round! We can round to powers of $1+\varepsilon$. This only multiplies each job p_j by at most $1+\varepsilon$ factor, thus the optimal value changes by only at most $1+\varepsilon$ factor.

Problem: we might have a geometrically decreasing series of sizes. If we try to round then nothing really changes. But, remember: small jobs don't matter. We just need to solve big jobs optimally.

Formalize. Given target T. We call large jobs the ones that have $p_j \geq \varepsilon T$. Round large jobs to $\varepsilon T, \varepsilon(1+\varepsilon)T, \varepsilon(1+\varepsilon)^2 T, \ldots, T$. There are $\log_{1+\varepsilon} \frac{1}{\varepsilon} \approx \frac{1}{\varepsilon} \ln \frac{1}{\varepsilon}$ such values. For any constant ε, the number of job sizes k is constant. So, we can solve in $n^{O\left(\frac{1}{\varepsilon} \ln \frac{1}{\varepsilon}\right)}$.

Consider the last task. If it is a small job that changes $C_{\max}$. Its size is $\leq \varepsilon T$. Before it was added, the machine was minimum and thus below average and thus below optimum T. So, $C_{\max} \leq T + p_{k+1} \leq T + \varepsilon T$. So, at worst we have $(1+\varepsilon)^2 \approx 1 + 2\varepsilon$ approximation.

§10.7 MAX-SNP problems

All polynomial approximation schemes boil down to enumeration. (Someone proved this.)

Max-SNP problems have some constant beyond which they cannot be approximated, meaning there is no polynomial approximation scheme. For some problems, there are amplifications: once you can show that they cannot be approximated to a constant, they can never be approximated to any constant. For example, the clique problem cannot be approximated. Not even to $n^{1-\varepsilon}$ (polynomial approximation!).

§10.8 Relaxation

Idea: if the problem is not solvable, we can change it to a solvable problem that is similar enough to the original problem. Then, we transform the solution back to the original problem. This is called relaxation because we usually remove or lessen certain constraints, making the solution space bigger. If we find a solution that is not feasible to the original problem, we have to move it back into the feasible space.

Problem 10.8.1 (Traveling Salesman Problem)**.** Given a graph with edge lengths. Find a path that visits each vertex exactly once.

Note that just finding the Hamiltonian cycle is already NP-hard. So, we will consider the **Metric TSP** instead: each vertex can be visited multiple times and they are separated by a metric that satisfies triangle inequality. On any graph, we can create a "metric completion" by finding APSP and defining edge lengths to those values instead. We will also consider only the undirected version too.

Also, note that if the path needs to visit a node more than once, by triangle inequality, you can shorten it so it visits each vertex once only.

To solve this problem, we will relax the path requirement: find the minimum spanning tree instead. (Any feasible TSP is a spanning tree plus on edge.) Thus, the optimal MST $\leq$ the optimal TSP. From an MST, we can generate an Euler tour which has distance $2\cdot$ MST, so we have a 2-approximation.

Note that metric TSP is MAX-SNP hard even if all edge lengths are 1 and 2 (and thus can't violate triangle inequality).

§10.8.1 Christofides's heuristic

Start with an MST. Note that we don't really need to traverse all the edges. We only need all vertices. The tour is Eulerian. So, we can just take an MST and add edges to make it Eulerian.

Problem: the odd vertices. To make them even, just add edges. Note that each edge fixes two odd vertices. So, we actually just need a minimum cost matching on odd vertices. The cost of the matching is at most the TSP.

In fact, the cost of matching is at most $\frac{1}{2}$ the TSP. Note that if we consider only the odd vertices, the cycle between them are two matchings combined. So, at least one matching is only at most half the TSP. Thus, we have $\frac{3}{2}$-approximate algorithm.

Caveat: we cannot just use min cost flow to solve this because this is not a directed bipartite matching.

Held-Karp came up with a way that might do $\frac{4}{3}$-approximation. Nobody has found a counterexample yet. Karlin-Klein-Oveis Gharian found $\frac{3}{2} - \varepsilon$ where $\varepsilon \geq 10^{-36}$.

§10.9 Scheduling problem with release dates

For $1 \mid\mid \sum C_j$, shortest processing time (SPT) heuristic is optimal. Proof by exchange argument: Suppose there is j such that $p_j > p_{j+1}$. Swapping them does not change other jobs' completion time except these two, allowing the sum to decrease.

For $1 \mid r_j \mid \sum C_j$ where r_j is the release date, this problem is much harder, because we cannot process short jobs because they are released. One idea is to schedule shortest available job greedily. However, this breaks in the case where we start with one huge job but a lot of smaller jobs become available later. We should not hastily schedule that one huge job.

An alternative problem is $1 \mid r_j, \text{pmtn} \mid \sum C_j$ where pmtn stands for preemption. A job can be paused at any time without losing any progress. A heuristic that runs optimally is the shortest remaining process time first heuristic (SRPT). This can be proved by exchange arguments based on the pieces of jobs running. If remaining time $p_j < p_k$ and both jobs are available, we swap around the pieces so all j pieces are finished before k.

Note that this alternate problem is a relaxation. So, first we find an optimum to SRPT, then we try to round. Suppose job j finishes at time C_j in the preemptive schedule, we insert job p_j starting at time C_j and expand the schedule (and leave old piece intervals blank). This gives us a feasible schedule because old C_j is obviously after job release time. Note that job j will finish in time $2 \cdot C_j$ with no further slow down because of SRPT order.

§10.10 LP Relaxation Technique

Consider the vertex cover problem. We can write it as an integer LP:

$$\min \left\{ \sum x_i, x_i + x_j \geq 1 \ \forall (i,j) \in E, x_i \in \{0,1\} \right\}.$$

(This polytope is called the stable set polytope. It has been shown that at every optimum value is 0, $\frac{1}{2}$, or 1. Any 0 is not in VC. Any 1 is in VC.)

ILP is NP-Hard, but if we remove the integrality constraint we have an easy LP problem. This is LP relaxation. Now, we have to figure out how to round to an integral solution.

We can round the usual way (to 0 if $< \frac{1}{2}$ and to 1 if $\geq \frac{1}{2}$). Notice that the constraints will still be maintained because if $x_i + x_j \geq 1$ then x_i or x_j is $\geq \frac{1}{2}$. Each x is at most doubled so we have 2-optimization. Note that this also works if each vertex has weight attached to it ($\min \sum w_i x_i$).

§10.10.1 Facility location problem

We would like to open facilities to serve clients. Each facility has opening cost f_i. Distance to client j is c_{ij}. Overall cost is $\sum_{\text{Opened}} f_i + \sum_{\text{Client}} s_j$ where s_j is distance from client j to the nearest facility.

ILP: Define $y_i = 1$ if open, $y_i = 0$ if not. Define $x_{ij} = 1$ if client j is assigned to facility i, 0 otherwise. Minimize $\sum y_i f_i + \sum c_{ij} x_{ij}$. For each client j, $\sum x_{ij} \geq 1$. For all i, j, $x_{ij} \leq y_i$.

We relax this by allowing $0 \leq x_i, y_i \leq 1$. Now, we round. The first step is called **filtering** where we try to remove small fractions that do not contribute to anything: Assign j to some nonzero x_{ij}. Write $C_j = \sum_i x_{ij} c_{ij}$ (average assignment cost for j). Some $x_{ij} > 0$ might have huge c_{ij}, but not all. We claim that at most $\frac{1}{\rho}$ total of x_{ij} is to facilities of assignment cost $> \rho C_j$. So, we zero out those large assignments. There remains a total of $1 - \frac{1}{\rho}$ assignment, so we fix by multiplying: $x'_{ij} = \frac{x_{ij}}{1-\frac{1}{p}}$, $y' = \frac{y_i}{1-\frac{1}{p}}$. The objective is increased by $\frac{1}{1-\frac{1}{\rho}} = \frac{\rho}{\rho-1}$. Now, every $x_{ij} > 0$ is to $c_{ij} \leq pC_j$.

Step 2 is facility opening. If y_i is small, the rounded solution pays f_i versus the fractional version pays $y_i f_i$. This gap is too large and won't give us a good approximation. Instead, we find a **cluster** of y_i whose total is ≥ 1. **Suppose our clients are in metric space.** We choose to open the cheapest facility in the cluster: $f_{\text{min}} = f \min \sum y_i \leq \sum f_i y_i$. Formally, take client with $\min C_j$. Consider all its facilities with $x_{ij} > 0$ (cluster). Set the cheapest to $y_i = \rho$ and others to 0. Assign every client with any nonzero x_{ij} to $i \in$ cluster to that open facility.

Cost of assigning client j' to i is $\leq c_{j'i'} + c_{ji'} + c_{ji} \leq 3\rho C_j$. Use $\rho = \frac{4}{3}$ to get 4-approximation.

§10.10.2 Randomized rounding

Problem 10.10.1 (Max-SAT)**.** Given a set of boolean variables and a set of clauses where each clause is an OR of a subset of variables or negations. The AND of this set is called a CNF formula (conjunctive normal form). Find an assignment that satisfies as many clauses as possible.

Naive solution: let us assign randomly where each $x_i = T$ with probability $\frac{1}{2}$. What is the $\mathbb{E}[\#(\text{satisfied clauses})] = \sum_{\text{clauses}} \Pr[\text{clause is satisfied}]$? For each clause, the probability of being satisfied is $1 - 2^{-k}$ where k is the number of variables in the clause. In the worst case, $k = 1$ so the probability is $\frac{1}{2}$. This gives us a 2-approximation (expected).

LP Rounding: Define variables

$$y_i = \begin{cases} 1 & \text{if } x_i = T \\ 0 & \text{otherwise} \end{cases}$$

and

$$z_j = \begin{cases} 1 & \text{if clause } j \text{ is satisfied} \\ 0 & \text{otherwise} \end{cases}.$$

The ILP is

$$\max \sum z_j$$

subject to

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

where C_j is the unnegated variables in clause j and C_j^{-1} is the negated vars.

As we solve the relaxed LP, we can round this randomly by treating each y_i as probabilities that $x_i = T$. Then, the term $\sum_{C_j} y_i + \sum_{C_j^{-1}} (1 - y_i)$ is the expected number of true variables in the clause.

We want to try to relate z_j to the probability that clause j is true because if we can do this, then summing up the probability gives us the number of clauses satisfied (as shown earlier).

Lemma 10.10.2

Let $\beta_k = 1 - \left(1 - \frac{1}{k}\right)^k$. (Note that this is increasing and the limit is $1 - \frac{1}{e}$.)

If a clause has k literals then Pr[clause j satisfied] $\ge \beta_k z_j$.

Proof. We consider a single clause j only, so assume w.l.o.g. that all variables in the clause are unnegated. Then,

$$Pr[\text{clause satisfied}] = 1 - \prod (1 - y_i).$$

This is minimized when the product is maximized subject to $\sum_{C_j} y_i = z_j$. The best we can do is to make y_i equal. So, the lower bound is $1 - \left(1 - \frac{z_j}{k}\right)^k$.

We claim that this is $\ge \beta_k z_j$. Note that the value $1 - \left(1 - \frac{z}{k}\right)^k$ at $z = 0$ is 0 and at $z = 1$ is β_k. The second derivative is < 0, meaning the function always curves down all the time. (Recall we said earlier that it is increasing.) Therefore, this value needs to be above $\beta_k z$ all the time (draw β_k vs z graph to see why). This means our probability is $\ge \beta_k z_j$.

Expected number of satisfied clauses is the sum of $\beta_{k_j} z_j \ge \sum \left(1 - \frac{1}{e}\right) z_j = \left(1 - \frac{1}{e}\right) OPT$. □

Note that an adversary can push us to the tight bound by making the clauses really large, so this our algorithm will do badly. However, recall that our simple algorithm earlier works well on large clauses. We should just use both algorithms and then pick whichever one gives a better answer.

Expected achieved value is $\geq \frac{1}{2}\left(\mathbb{E}[\#(\text{clauses satisfied by naive})] + \mathbb{E}[\#(\text{satisfied by LP})]\right)$. This is $= \frac{1}{2}\left(\sum \left(1 - 2^{-k_j}\right) z_j + \beta_{k_j} z_j\right)$ which is $\geq \frac{3}{4}$. This yields a $\frac{3}{4}$-approximation algorithm.

It is known that Max-SAT is Max-SNP-hard. For Max-3SAT, our naive algorithm achieves $\frac{7}{8}$-approximation. Doing $\frac{7}{8} - \varepsilon$ is NP-hard.

11 Parameterized complexity

The idea of NP-hardness is to say that the problem is hard over all instances. In practice, you are interested in some instances only. We can define the parameters that measure the hardness of that instance and give a runtime based on that parameter.

Consider the vertex cover problem. Suppose we know the optimum cover uses k nodes. Then, we can simply try all k-subsets of vertices. So, runtime is $O\left(mn^k\right)$.

A better approach is "bounded search tree" method. We consider the solution space as a search tree then argue that the tree is shallow. Pick an edge. We know one end is in the VC. Let us try each one and recurse to find the rest of VC. Note that this is a degree 2 search tree, and we only need to go k levels deep. Runtime is $O\left(2^k m\right)$. This runtime is called fixed-paramter tractable runtime: It can be expressed as $f(k) \cdot \mathrm{poly}(m, n)$.

Note 11.0.1. I notice that the "picking arbitrary edge" part corresponds to picking a constraint. That might help me come up with other fixed-parameter algorithms in the homework.

§11.1 Kernalization

In polynomial time independent of k, you can find a hard "kernel" of the problem of size $f(k)$ independent of n. Solve kernel in time $g(k)$.

Consider the vertex cover problem again. Any vertex of degree $> k$ must be in OPT, otherwise you must use all k neighbors. Mark all these nodes and remove incident edges. The new graph has degree $\leq k$ and still has a vertex cover of size $\leq k$. The graph has at most k^2 edges and k interesting vertices. We can use the previous algorithm to solve in $O\left(k^2 \cdot 2^k\right)$. So, overall runtime is $O\left(m + k^2 \cdot 2^k\right)$.

§11.2 Treewidth

Idea: use recursion on vertex to solve graph problems. The eliminated vertex generally creates some hidden dependencies between the neighbors of that vertex. Represent those dependencies by adding edges between neighbors. If repeated remove vertices and add dependency edge, this yields an "elimination ordering" for the graph. You can answer the problem by considering how vertex interacts with its neighbor. The treewidth is the maximum degree we may encounter. For a tree, the treewidth is one because we can repeatedly eliminate the leaves (with degree 1) without introducing dependencies.

Graphs with treewidth 2 are called series-parallel graphs. Many problems are tractable for small treewidth, e.g. SAT.

For SAT, we think of each variable as a vertex. Create an edge if two vertices share a clause in the formula. Solve by elimination in runtime 2^T where T is the treewidth. Note that each clause has a set of satisfying assignments for the variables in that clause and the problem is to pick an assignment that is consistent with other clauses' assignments.

Take a variable x. Consider all clauses it contains. Find an assignment to the other variables that also work for x. Define combined clause from all clauses containing x. List its satisfying assignments. The size of this clause is = no. of neighbors of $x \leq$ treewidth. Recurse. Runtime is $2^{\text{max clause size}}$.

12 Computational Geometry

For a computational geometry problem, we consider points, lines, and planes to be primitives. Operations intersect, compare lengths, and finding angles are considered to be tractable in $O(1)$ time. (We will only worry about precision and other things when we actually implement the algorithm.)

The key idea is to do a recursion on dimension. We take a d-dimensional problem, do work to get $d-1$ dimensional problem and recurse.

Problem 12.0.1 (Orthogonal range queries)**.** Given a set of points in d dimensions. Set up a data structure that allows querying "which points are in this given box."

For 1-D case, we can just create a BST, augmented with min/max values, and we just have to traverse in between (or sort then binary search for range). Runtime is $O(\log n+k)$ is we need to output k points. If we just want to count or check for emptiness, with augmentation on BST we can solve in $O(\log n)$.

To generalize, we can solve each dimension separately. Problem: we have to intersect a set of 1-D answers. Set intersection is expensive, e.g. querying one dimension in full and another dimension very small takes linear time to find empty output.

Let us think about 2-D case. Suppose we already know the x interval. If we have built a BST on y coordinates of these points, then we can query quickly. We can just precompute the BSTs for each interval that starts and ends on existing points. In d dimensions, the runtime would be $O\left(d \log n + k\right)$.

Two problems: Building time is huge. Space is huge (n^{2d-1} in dimension d). Instead of enumerating the vertices, we will use the idea of augmentation instead. First, we build BST on x coordinates as planned before. Each subtree defines a subinterval of x axis. When we query an interval, we just need to union $O\left(\log n\right)$ of them (argument similar to segment tree). So, we build y-BST for points in each x subtree. To answer a query, we need to find at most $O(\log n)$ y-BST, run $O(\log n + k)$ query on each, and then union the answers. Overall runtime is $O(\log^d n)$ for emptinss and counting queries.

Analyze the size. There are n tree nodes in x dimension. Each point is in exactly one y search tree per each level of x tree. So, size is $O(n \log n)$. Generally, in d dimensions space is $O(n \log^{d-1} n)$ space. Runtime is $O(\log^d n)$. There is an idea of **fractional cascading** that can reduce the runtime to $O(d \log n)$.

§12.1 Sweep Algorithms

We think of the x dimension as time dimension instead. So, for 2-D problems, we are solving them as if they are time-variant 1-D problem. For 3-D and above, we can do recursion.

Problem 12.1.1 (Convex Hull)**.** Given a set of points, find the smallest containing convex polygon. The hull can be described by listing the points in hull order (clockwise or counter-clockwise).

There are 70+ algorithms for 2-D convex hull, each of which demonstrates a different technique. Here, we are concerned with an algorithm that finds the upper hull using sweep line technique. First, sort the points by x coordinates (time). Sweep a line from left to right, tracking what the upper hull looks like so far using stack. The invariant is we should always be turning clockwise. We can do the same for lower hull, giving us a full hull in $O(n \log n)$ time: $O(n \log n)$ for sorting and $O(n)$ for keeping stack.

In a sense, the convex hull problem is equivalent to the sorting problem and you can reduce from one to another. For some inputs, there are many points but the final convex hull is very simple. **Output-sensitive algorithms** can solve the convex hull problem with time complexity dependent on the output size. For example, Chan '96 solved it in $O(n \log k)$ where k is the number of points on the hull.

Problem 12.1.2 (Half-space intersection)**.** Given a set of half-spaces. Draw the intersection.

Note the duality: there is a mapping between points on a line and lines through a point. Point (a, b) maps to and from line $L_{ab} = \{(x, y) \mid ax + by = -1\}$. So, half-space intersection is equivalent to convex hull.

LP in low dimension has strongly polynomial based on computational geometry techniques. Specifically, it is solvable with time linear to the number of constraints and exponential in the number of dimensions. For example, for 2-D cases, you can simply do half-space intersection and try out each vertex.

Another interesetnig problem is point location problem which can be solved using persistent search trees.

§12.2 Voronoi Diagrams

Problem 12.2.1. Given a set of points p_i in the plane. We want to answer a query: given point q, find the nearest p_i.

Definition 12.2.2 (Voronoi regions)**.** $V(p_i)$ = the set points in plane closer to p_i than all other points.

For one point, the voronoi region is just the entire plane. For two points, the voronoi regions are the half-planes with the boundary being the perpendicular bisector through the line through two points. For three points, draw three pairwise perpendicular bisectors to get six regions, and assign two closest to each point as if each bisector is cut down in half. The point where three half-bisectors meet is called the "voronoi point," which is the circumcenter of the triangle and is equidistant from the three points. Note the trend as we add more points: bisectors are being cut and we start creating closed polygons.

Each voronoi region is a convex polygon containing one p_i, so there is exactly n regions. Assume non-degeneracy: no four points lie on a circle. (Each voronoi point is the intersection of three bisectors.)

The nearest neighbor problem can be reduced to figuring out which voronoi region a point q is in, which we know how to solve. To analyze this, we have to be able to answer what is the size of vornoi diagram (VD) on n points? By size we mean the number of line segments.

Note that VD is a planar graph because the edges do not cross. (We add a "point at infinity' to be second endpoints for infinite lines.) By euler's formula, $V - E + F = 2$, which can be proved by induction.

Proof. Remove each edge where both ends have degree ≥ 3. E decreases by one but F also decreases by one. For each degree 2 vertex, remove it and combine the two edges. V decreases by one but E decreases by one. Edge on degree one vertex and the vertex can be removed without changing the number of faces. Finally, we go down to the case where $V - E + F = 2$. This sum never changes. $\square$

For our VD, $F = n$. V = number of voronoi points, which we have already argued have degree 3 each. Each edge has 2 voronoi endpoints. Let us do double counting. $2E$ counts the two ends of each edge. $2E$ is equal to the sum of degrees which is $\geq 3(V+1)$ (1 is point at infinity. Point at infinity might have > 3 degree.). So, $2(n - V - 2) \geq 3(V+1)$, so $V \leq 2n - 7$. The number of voronoi points and the number of edges are linear in the number of points.

The only thing left is to actually construct the VD. We will use sweep line algorithm. Sweep from top to bottom and build VD at sweep line. Problem: the VD is not fully determined by what we see above the sweep line. For example, if we have three points but we have only sweeped through two lines, when we meet the third point our bisector might need to shrink. However, note that VD area that is close to an input point above the sweep line than it is to the sweep line itself will not change. That area is a parabola! (The sweep line is the directrix. The input points at the bottom, also called the "sites," are the input points.) Fortune's algorithm find the boundary pieces of parabolas, called the "beach lines." As the sweep line descends, we trace out the VD behind the beach line.

Let us look at one point (x_f, y_f) and the sweep line $y = t$. Parabola is $(x-x_f)^2+(y-y_f)^2 = (y-t)^2$. Fix x and differentiate. We have $\frac{dy}{dt} = \frac{y-t}{y_f-t}$. If we have several parabolas at a particular fixed x, then the lowest focus of these parabolas descends fastest.

When our sweep line hits a point ("site event"), we create a degenerate parabola, then as we move down that parabola widens. This might result in joining the beach lines.

(The dual of VD is Delaunay triangulation problem.)

I got lost here. Please use Wikipedia or something. :(

13 Online Algorithms

Until now, we have been dealing with problems where we can look at the entire input and then output the answer after. In this chapter, we will learn about problems which require immediate output after each input. Some example problems are the stock market and paging.

Let the input sequence be $\sigma = \sigma_1, \sigma_2, \sigma_3, \ldots$. After each input, produce output or perform action. Optimize cost of outputs/actions: $c_{\text{min}}(\sigma)$. It is often easy to compute $c_{\text{min}}(\sigma)$ given σ.

§13.1 Ski-rental problem

Problem 13.1.1 (Ski-rental problem)**.** Let renting skis costs 1 unit. Buying ski costs T units. On one day, you might suddenly decide that you hate skiing. When should you buy?

"The idea that the past predicts the future is very optimistic. There is a guy named Murphy around: The moment that you buy ski, you'll fall and break your leg." - David Karger

How can we assess the quality of an online algorithm? We compare it to the best possible given σ.

Definition 13.1.2 (k-competitiveness)**.** Algorithm is k-competitive on σ if $C_A(\sigma) \leq kC_{\text{min}}(\sigma)$.

Algorithm is asymptotically k-competitive on σ if $C_A(\sigma) \leq kC_{\text{min}}(\sigma) + O(1)$. k here is called the competitive ratio.

Algorithm is k-competitive overall if it is k-competitive for all σ.

Suppose we always rent. Then, we have $\frac{\infty}{T}$ worst case ratio, which is non-competitive. If we buy immediately, we have $\frac{T}{1}$ worst case which is T-competitive.

If we rent d days then buy on day $d+1$, let us find the worst σ. Think about the adversary. Whenever they rent and we rent, the ratio is 1 and does not improve. When the adversary buys and we keep renting, our ratio starts getting worse. When we buy, our cost remains fixed but the adversary might keep paying more, meaning the ratio gets better. This means the worst ratio is calculated at the moment when we buy. The ratio is $\frac{d+T}{\min\{d+1,T\}}$. If $d+1 \leq T$, then ratio is $\frac{d+T}{T}$ which is minimized when $d = T$. When $d+1 \leq T$, set $d = T$. In both cases, the ratio is ≤ 2. Thus, we achieve competitive ratio of 2.

§13.2 Selling stocks

§13.2.1 No splitting

Suppose we have bought a stock on day 0. On each day $1, 2, \ldots$, we have to decide whether to sell or to keep it. Note that if there is nothing constraining the market, we cannot ever be competitive. For example, if we choose to sell on day 1, on day 2 price might go up to infinity. If we do not sell on day 1, on day 2 price might drop down to zero. So, in our problem we will assume we know the duration, the max price M, and the min price m. Define $\phi = \frac{M}{m}$.

Because of the constraints, at the very least we can sell at price m, giving us the competitive ratio of ϕ (because the optimal solution might sell at $M - 1$).

To solve this better, we define "reserve price" to be $\sqrt{M \cdot m}$. This means we sell the first time it gets larger than the reserve price or at m if we never reach this. There are two possible cases. 1) There is $\sqrt{M \cdot m}$ offer and we take it but the optimum is M then we get ratio of $\sqrt{\frac{1}{\phi}}$. 2) The best offer is $< \sqrt{M \cdot m}$ and we sell at m, so we get ratio $\sqrt{\frac{1}{\phi}}$. So, this is $\sqrt{\phi}$-competitive.

§13.2.2 With splitting

Now, suppose we can sell our stock in fractional pieces. We can use multiple reserve prices, e.g. sell the first half at $m^{\frac{1}{3}}M^{\frac{2}{3}}$ and the second half at $m^{\frac{2}{3}}M^{\frac{1}{3}}$. In this example, we find by case analysis that at least half of our stock is sold within $\phi^{\frac{1}{3}}$ of the optimum, giving us $\frac{1}{2}\sqrt[3]{\phi}$-competitive.

To generalize, we set reserve prices at $m, 2m, 4m, \ldots, M$. There are $\log \phi$ such prices. We sell $\frac{1}{\log \phi}$ fraction at each reserve price. One of the successful reserve prices is at least $\frac{1}{2}OPT$. This means we have $2 \log \phi$-competitive ratio.

§13.2.3 Using randomization

Suppose we cannot split our stock. We can set random reserve price from $m, 2m, 4m, \ldots, M$. There is $\frac{1}{\log \phi}$ chance that we choose the price r giving $\frac{1}{2}OPT \leq r \leq OPT$. If so, we earn $\frac{OPT}{2}$. So, the expected earning is $\frac{1}{\log \phi} \cdot \frac{OPT}{2} = \frac{OPT}{2 \log \phi}$. We have $2 \log P$ competitive ratio in expectation.

§13.3 Scheduling problem

Suppose jobs arrive over time and we have to assign to machines when they arrive.

For $P \mid\mid C_{\max}$, greedy Graham's rule from the 60s which assigns arriving jobs to the currently least loaded machine gives competitive ratio of 2. We already proved this in the unit of approximation algorithms.

In 1995, Bartal et al. improved the ratio to $2 - \frac{1}{70}$. Recall that the worst case for Graham's rule is when we have filled up all machines with tiny jobs and then suddenly one huge job which we have not readied a spot for comes in. Bartal prevents this problem by splitting the machines into two groups: slightly underutilized, and slightly overutilized. As a huge machine comes, we can change the group threshold. With careful balancing, Bartal got $2 - \frac{1}{70} \approx 1.986$.

In 1996, Karger splits the machines into different groups instead, constructing a linear program to optimize the group sizes. The ratio went down to 1.945. In 1997, it went down to 1.923. There is a known lower bound of $\frac{4}{3}$.

§13.4 Paging problem

In computer memory system, some spaces are fast but small and some are slow but large. We have to decide where to store each information and move around as needed. Because slow memory tends to be extremely slower than fast memory, we divide the memory into pages and whenever we need something on a page we fetch the whole page instead of just one byte of data. Basically, we are relying on "locality of reference": when we access information, we will probably continue accessing the information near it.

Each time the memory access is unavailable in the fast memory, we have what is called a "page fault." We have to "evict" some other page to make room in the fast memory. One idea is to evict the least frequently used. Another is to evict the least recently used. Another is to evict FIFO or LIFO. If we can predict the future, the optimal solution is to evict the one that will be used furthest in the future (Belady's algorithm).

It turns out that least frequently used and LIFO are not competitive. Least recently used and FIFO are k-competitive where k is the number of pages that can fit in the fast memory.

§13.4.1 Least recently used (LRU)

Suppose we break any sequence of page requests into "phases" of $k + 1$ faults generated by our algorithm. If there are k distinct pages in a phase, then none of them become old enough to be evicted. So, a phase must have $k + 1$ distinct pages. This means OPT has at least one fault. Therefore, we get the ratio of $k + 1$.

We can refine this to k. Divide requests into phases of k distinct page requests. In each phase, each distinct request causes at most one fault because none of them can become the $(k+1)$-st oldest and be evicted again. Total is at most k faults per phase.

Now, we want to analyze the optimal solution and show that there is at least one fault per phase to obtain k-competitive ratio. The idea is to move first item of each phase to previous phase. (First phase now has $k+1$ requests. Other phases has k requests.) At start of phase 2, a_{k+1} is in memory. $a_{k+1}, \dots, a_{2k}$ include k distinct requests (by previous definition). Suppose there is no fault after a_{k+1} is fetched until a_{2k}. At a_{2k} we know exactly which pages are in memory. By definition, a_{2k+1} is distinct from them and thus causes a fault. If there is a fault after a_{k+1} but before a_{2k+1}, that is also at least one fault. Therefore, in both cases, we have k-competitive ratio.

This is a tight bound. We can give an adversarial case to demonstrate this. Suppose we have a set of $k+1$ pages and run the online algorithm. At each step, we will request a missing page so we miss every time. We have to also show that the offline algorithm does well. The offline algorithm knows the sequence, so on a miss it knows which one is the best to evict. The next $k-1$ requests will be hits. OPT misses only once per k requests, so it is doing k times better. Therefore, k is the exact bound we can achieve.

Note that this is proved by Sleator and Tarjan... again!

In modern machine, $k \approx 10^3$ or 10^4, so k-competitive is bad. An alternate way to analyze these processes is to use probabilistic methods, however, this is quite optimistic because real life programs have complex but predictable behaviors. There is also a concept called "resource augmentation": if we have k pages but OPT has $h \leq k$ pages, then LRU is $\frac{k}{k-h+1}$-competitive. Basically, we are comparing online algorithm with additional resource versus handicapped offline algorithm. If $h = \frac{k}{2}$, then we have roughly 2-competitive which is quite good.

§13.5 Preventing predictability via randomization

Note that so far we have simulated the online algorithm then create an adversary to design a bad sequence based on the result we get. So, one way to prevent these tight lower bounds is to use randomization.

Randomized online algorithms can be characterized in two ways: 1) as algorithms that flip coins while running, and 2) as a probability distribution over deterministic algorithms. Basically, we can flip the coins in advance and use the determined bitstring.

Possible adversarial models are: 1) oblivious, 2) fully adaptive (knows the coin flips), 3) adaptive (knows only the coin tosses until present but not the future).

We will consider the oblivious adversary for paging problem. Fiat et al. came up with an algorithm that combines random eviction with LRU which is called "marking algorithm" M. Initially, all pages are marked. On each fault, if all pages are marked, we unmark all pages, evict a random unmarked page, then fetch the requested page. Finally, whether we have a fault, we mark the requested page. Fiat proved that this algorithm is $O(\log k)$-competitive.

Our proof is based on phases again. A phase will start at the first fault and restart on the $(k+1)$-st request. There are k distinct requests per phase. At the start of each phase is when all pages are already marked and then unmarked. So, we might as well think of the cost of the algorithm as just the sum of cost of each phase. Each phase is defined by input, not influenced by random choices, but mark bits match phases.

Observe that each item, once marked, stays in the memory at least until phase ends (because it never gets booted until it is unmarked at the end). So, each page causes at most one fault in that phase. This means we might as well ignore all except the first request for page in a phase. (This won't hurt our algorithm but might do to the adversary.)

We will say that phase i starts with M (our algorithm) having pages set of pages S_i in memory. Define: request is clean in phase i if it is not in S_i. By definition, M faults on this request (because it is not in the memory when we need it). We claim that stuff that are not in S_i is old, so OPT doesn't have it as well and thus misses. Define: request is stale in phase i if it was in S_i. M will fault only if it evicted this page from S_i before it gets requested here. We will show that this is unlikely due to randomization.

Suppose that in a phase, there has been s stale and c clean requests so far (considering the distinct pages only because we said earlier don't need to care about repeated pages). They all got marked and are still in memory. So, $s+c$ pages are known to be in the memory. Consider the next stale request. The probability of faulting here is equal to the probability that we had evicted this page before. We know each of the c clean requests evicts an unmarked page from S_i. Stale requests (were in S_i), if evicted, are also brought back by requests and thus are in memory now. The rest of S_i, totaling up to k pages, are candidates for what might be missing. So, c of $k-s$ candidates from S_i are missing. By symmetry, each page is missing with probability $\frac{c}{k-s}$.

We can analyze the whole phase. Suppose there are c_i clean requests and $k-c_i$ stale requests. We pay c_i for each clean request. For stale requests, we miss with probability $\leq \frac{c_i}{k} + \frac{c_i}{k-1} + \dots \frac{c_i}{k-(k-c_i-1)} = c_i\left(\frac{1}{k} + \frac{1}{k-1} + \dots + \frac{1}{c_i+1}\right) = O(c_i \log k)$ in the worst case.

Now, let us analyze OPT. We can't really compare phase by phase, but we can use potential function. Define $\Phi_i =$ the number of pages that are different in cache positions of M and OPT. $0 \leq \Phi_i \leq k$ obviously. Suppose we got c_i clean requests. They are not in S_i. At least $c_i - \Phi_i$ are not in OPT's cache at start of i.

At the end of round M we have k requests. OPT is missing Φ_i of most recent requests. So, they were evicted during this phase. OPT had this many page faults. So $OPT \geq \max\{c_i - \Phi_i, \Phi_{i+1}\} \geq \frac{c_i - \Phi_i + \Phi_{i+1}}{2}$ (because max is at least the average). The sum telescopes to $\Phi_f + \sum \frac{c_i}{2} - \phi_0$. So, $(\log k)$-competitive.

§13.6 Finding lower bound for randomized online algorithms

We view the problem as a game between the algorithm designer (A) and the input adversary (a). The score is $C_A(\sigma) - kC_{OPT}(\sigma)$. If < 0, we have achieved k-competitiveness ($C_A(\sigma) \leq k \cdot C_{OPT}(\sigma)$), otherwise we failed. Algorithm designer wants to maximize this. Input adversary wants to maximize. This is a zero-sum two-player game.

Recall the homework problem: If one player's mixed strategy is known, another player can just set 1 to the worst case that would happen. Our randomized algorithm has a probability distribution over A. We want our algorithm to $\min_{\text{random}A} \max_\sigma \mathbb{E}[C_A(\sigma) - k \cdot C_{OPT}(\sigma)]$. By equilibrium, we know we can just switch the orders. We can view this as if the adversary chooses a random distribution over input and our single algorithm solves it purely.

Theorem 13.6.1 (Yao's Minimax Theorem)

Best competitive ratio achievable by a randomized online algorithm (whose distribution but not choices are known to the adversary) is equal to the best competitive ratio achievable by a deterministic online algorithm versus a known distribution over inputs σ.

Corollary 13.6.2

If there exists a distribution over σ such that no deterministic algorithm A can do well, then no randomized algorithm A can do well against all deterministic inputs.

So, to find a randomized online algorithm competitive ratio lower bound, just come up with a bad distribution of inputs and show that no deterministic algorithm have expected ratio r. Then, we can conclude that no randomized algorithms can beat ratio r.

§13.6.1 Lower bound for paging algorithms

Here is one bad input distribution we could use: uniform distribution over $k+1$ pages.

For an online deterministic algorithm on this sequence, regardless of whatever is in the memory, $\Pr[\text{fault}] = Pr[\text{wrong page requested}] = \frac{1}{k+1}$. Every deterministic algorithm has $\mathbb{E}[\text{page faults}] = \frac{1}{k+1}$ per sequence.

When OPT faults, it evicts the page that is requested again furthest in the future. It won't fault again until the request arrives. When? That sequence is the last of $k+1$ pages. There are no faults until every of the $k+1$ pages show up. This is the coupon collector's problem! So, it takes $O(k \log k)$ requests in expectation before we see a fault.

Our deterministic algorithm faults with probability $\frac{1}{k+1}$, so about $k+1$ requests in expectation per fault. So, the deterministic algorithm has $O(\log k)$ times as many faults as OPT. Thus, any randomized online algorithm can be at best $O(\log k)$-competitive.

§13.7 k-server problem

Manasse et al. (1988) came up with a problem called k-server problem which is the generalization of many other problems.

Problem 13.7.1 (k-server problem)**.** Given a metric space and k servers that move between points. Once we receive a request which is a point in space, we must move some server to the point. The cost is the total distance traveled by all the servers combined.

This problem covers, say, assigning repairmen. It does not cover taxi services because taxi has to move to certain destination after each request. Interestingly enough this problem also covers paging. Consider the uniform metric. A server is a slot in the memory. Moving a server to a point is equivalent to evicting and receiving a page. This problem also covers weighted paging, multi-head disk drives, etc.

Consider the case with three points. A and B are near. C is very far from A and B. We have servers on A and C. Let the requests be $A, B, A, B, \ldots$. Then, greedy solution would have the first server alternate around, resulting in infinite cost in the limit. Meanwhile, the optimal strategy is to send the server from C to B and then you would not have to move ever again.

Yet, we can still gain some insight from this. This is very similar to the ski-rental problem where after a certain point, we pay a large cost to just settle things down.

The harmonic algorithm moves a server with probability proportional to $\frac{1}{\text{distance to request}}$. Fiat showed that this algorithm is $O(k^k)$-competitive.

The work function algorithm tracks what the offline algorithm would've done until now, then tries to have the online algorithm converge to the current offline algorithm. Define $W_i(X)$ (the work function) to be the optimal cost of serving the first i requests and finishing with server on set of points X. At each step, online algorithm chooses its X_i to optimize the $\min_X W_i(X) + d(X_{i-1}, X_i)$. Computing the offline optimum might be NP (not sure?). However, work function algorithm is $(2k)$-competitive. Very good!

§13.7.1 1-D case: Double-coverage algorithm

Suppose we have servers spread on a line. For each request, we should only consider the adjacent servers and move one of them. We do not know which one should move to that point, so one idea is to just move **both** there until one reaches it! (This doesn't really help anything but it makes the analysis simpler.) Locally, we are spending at most twice the optimum. We aren't sure about globally because the servers might not be in the same starting position as ours in OPT. We will prove this is k-competitive.

We use potential function to analyze these kind of online algorithms because we usually care about the amount of work done over a series of requests, not just a single request. Our potential function should measure how different DC (our algorithm) differs in positions from OPT. We look at the minimum cost to move DC servers to OPT, which is just a min-cost matching between DC and OPT server positions. Let M be the value of this matching. We also define d to be the distance between servers in DC. Then, we define:

$$\Phi = kM + \sum_{i<j} d(s_i, s_j).$$

In our analysis, we will first move OPT and update Φ, then move DC then update Φ. Show that if OPT moves distane d, Φ increases by $\leq kd$. If DC moves d, Φ decreases by $\geq d$. If we can show these two things, then we have asymptotic k-competitive algorithm because if OPT moves D and DC moves $\geq kD$ then the final Φ decreases.

Suppose OPT moves by d. The term kM is the only term that changes (because the other term is for DC only). At worst, OPT moves away from its match by d, so M increases by d, meaning potential is incremented by kd.

Suppose D moves by d (already moved OPT). (Note that this d is different from the OPT case.) Case 1: the request is outside the range of DC servers. We move the closest server to the optimum. We know one of the OPT servers is already there (because we already moved). So, M decreases by d, meaning kM term decreases by kd. Why? We can assume w.l.o.g. that in the matching it is the rightmost servers of DC and OPT that are matched. Suppose that the moving DC is matched to a different OPT server. We can just swap and the matching cost would be have been less (proof by picture).

We also have to analyze the sum term, not just kM. Since we are moving the outermost server, its distance to all other servers increase by d each. So, the sum increases by $(k-1)d$. Therefore, total Φ change is $-d$.

Case 2: request R is between 2 DC servers. For the M term: Assume w.l.o.g. one of the moving servers is matched to the OPT at R (same non-crossing argument). The server matched to OPT moves d, so decreases M by d. The other server might be moving away and increase M by at most d. So, $\Delta M \leq 0$. Now, for the sum term, note that the two points move in opposite direction. The center of gravity is unchanged, so the average distance to other points is still unchanged. Everything cancels except the $d(s_i, s_j)$ term themselves which decrease by $2d$. So, total Φ decrease is $\geq 2d$. We have completed the proof.

It turns out this argument also generalizes to trees. We can just move every server that is not "blocked" by different servers to the request (more than 2). By the same analysis, we will get k-competitive.

Why are trees so special? There is a problem called "metric embedding" where you try to represent general metrics as combinations of special metrics. It was proven that any metric is a sum of $\log^2 n$ tree metrics. This leads to a randomized k-server algorithm in $O\left(\log^2 n \log^2 k\right)$-competitive where n is the number of space points and k is the number of servers.

Can we do better? From paging, we know the lower bound for deterministic algorithm is k and for randomized is $\log k$.

14 External memory algorithms

In computers, RAM is fast but accessing disk is so slow that the algorithmic runtime is just determined by the number of disk accesses. In the domain of external memory algorithms, we consider the algorithms that minimize the number of disk accesses.

Parameters for such algorithms are:

- Block size B: the number of items that fit in a block.
- Problem size N: the number of items in input.
- Memory size M: the number of items that fit in fast memory.

§14.1 Basic algorithms

Problem 14.1.1. Given an array on disk, scan to find the desired item.

N items to fit in $\frac{N}{B}$ blocks. Read each block at once. So, runtime is $O\left(\frac{N}{B}\right)$.

Problem 14.1.2. Reverse the array.

Runtime is also $O(\frac{N}{B})$: read each lock, reverse, write.

Problem 14.1.3. Given two $n \times n$ matrices. Add them.

Now, this isn't easy. How do we store matrices on the disk? One way is the "row-major order" which basically divide each row into blocks. Put the blocks from the top row first to the bottom row. We can just add the corresponding blocks. So, runtime is $O\left(\frac{n^2}{B}\right)$.

Problem 14.1.4. Given two $n \times n$ matrices. Multiply them.

If we do simple matrix multiplication where we have to read the row from A and column from B then multiply to get one value (out of total n^2 values), we need $\frac{n}{B}$ blocks for A rows and n blocks for B columns (because B is in row-major order). So, runtime is $O(n^3)$ (n rows of output, each row requires $\frac{n}{B}$ reads from A and each cell requires n reads from B).

What we can do is to store B in column-major order instead. (The problem is how to transpose.) Reading B columns take $\frac{n}{B}$ time instead. So, runtime is $O\left(\frac{n^3}{B}\right)$.

This is still wasteful! Many outputs depend on the same input row but different columns and vice versa. We can use a different layout called the **block layout**. For example, we can break the matrix multiplication AB into

$$\begin{pmatrix} A_1 & A_2 & A_3 \\ A_4 & A_5 & A_6 \\ A_7 & A_8 & A_9 \end{pmatrix} \begin{pmatrix} B_1 & B_2 & B_3 \\ B_4 & B_5 & B_6 \\ B_7 & B_8 & B_9 \end{pmatrix} = \begin{pmatrix} A_1B_1 + A_2B_4 + A_3B_7 & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{pmatrix}.$$

We want each block to be as large as possible but still retain fast computation speed. So, each submatrix should fit into memory: size $\sqrt{M} \times \sqrt{M}$.

For one multiplication: We read the entire block in A and B, multiply them in memory, and write back. This takes $\frac{M}{B}$ block operations.

We have $\frac{n}{\sqrt{M}}$ blocks per row/column to produce one $\sqrt{M} \times \sqrt{M}$ output submatrix. There are $\left(\frac{n}{\sqrt{M}}\right) = \frac{n^2}{M}$ output submatrices. This means we need $\frac{n^3}{M^{3/2}}$ submatrix product operations, each taking $\frac{M}{B}$ time. So, total runtime is $O\left(\frac{M^3}{B\sqrt{M}}\right)$.

Note that this is based on the naive $O(n^3)$ sequential algorithm. An algorithm by Strassen runs in $O(n^{2.763})$. We can externalize this the same way to get a faster external memory matrix multiplication algorithm.

§14.2 Linked lists

We want to store a linked list in external memory. Operations to support are

- Insert/delete at current position (pointer)
- Scan/find next

Naive solution: For insert/delete, we need to read the block at the pointer. Also read prev/next. Delete them from memory. Write new values. This takes $O(1)$ time. For scan, we need $O(1)$ per step.

Note that scan is excessive. If the elements are stored contiguously, we could do $O\left(\frac{1}{B}\right)$ amortized per item. However, this messes with insertion because the boundary items might be moved into other blocks. Let us think of a way to solve this problem.

For deletion, if we simply do not rearrange the blocks, leaving a hole, we might have one item per block, giving $O(1)$ time per item for scan again. So, we only allow the number of items per block to go as low as $\frac{B}{2}$ items/block. When we drop below this, we combine with the next block. Case 1: the next block has $\leq \frac{B}{2}$ elements. We can simply combine. Case 2: next block has $> \frac{B}{2}$ elements. So, we have more than B elements. What we can do is rebalance into two blocks of $\geq \frac{B}{2}$ elements. Therefore, deletion (still) takes $O(1)$ to delete, but scan is now $O\left(\frac{1}{B}\right)$.

For insertion, if there is a hole, we use it. If there is no hole, we need to split a block into two blocks of $\geq \frac{B}{2}$ items each. Therefore, insert also takes $O(1)$.

§14.3 Search trees

We can move binary search tree into external memory. We need $O(\log N)$ reads to look up and $O(\log N)$ writes to insert/delete. Because we are dealing with external memory, however, comparison runtime doesn't really factor into our analysis at all. What we are aiming for is minimizing the number of block reads/writes. So, we need to find a way to store the tree efficiently.

One idea is to store parts of trees with B items each to get locality of reference. Each part would have height $\log B$. One read implies $\log B$ progress. So, we only need $\frac{\log N}{\log B}$ reads.

§14.3.1 B-trees

Let us talk about B-trees: trees of degree B. The depth of the tree is $\log_B N = \frac{\log N}{\log B}$. We will maintain these invariants:

- All leaves are at the same depth ($\log_B N$).
- Leaves are mostly full.
- Keys are at leaves.
- Copy some keys up from leaves to serve as splitters on internal nodes.

To scan the tree, simply scan the leaves. To search, we need to arrive via internal nodes. How do we insert/delete while staying balanced and maintaining invariants?

For insertion, we have a problem when we insert into a full leaf block. We can split into two leaf blocks. Copy middle key up to the parent as splitter. If the parent is full, then we also recurse. If we reach the root and need to split, we simply add another layer on top of our tree (a new root). This involves $O(\log_B N)$ work.

For deletion, we might get almost empty leaves. So, we will try to maintain $\geq \frac{B}{2}$ items per node. This can be done by merging with a neighbor block and delete the splitter from the parent recursively.

B-trees are interesting because they maintain balance by allowing degrees to vary between $\frac{B}{2}$ and B. Meanwhile, binary search tree keeps the degrees relatively fixed and maintain balance through rotations.

§14.4 Sorting

We can use quicksort and try to change it into an external variant. Naively, the runtime is $O\left(\frac{N}{B}\log\frac{N}{B}\right)$.

Can we do better? If we sort using a B-tree, the runtime is $O\left(N\log_B N\right) = O\left(\frac{N}{\log B}\log N\right)$ which is much worse. What about merge sort? Merge sort has $O\left(\log N\right)$ merge phases. Each merge phase is simply a scan. Runtime is $O\left(\frac{N}{B}\log\frac{N}{B}\right)$, same as quicksort.

Observe that M is missing from our runtime. We could make use of our fast memory. Notice that during merge sort if we have $\leq M$ items, we can scan and sort all in memory in $\frac{M}{B}$ time. So, runtime is $O\left(\frac{N}{B}\log\frac{N}{M}\right)$.

Idea: do $\frac{M}{B}$-way merge instead of 2-way merge. To merge $\frac{M}{B}$ sorted lists into one list with one scan pass, we put the front of each list in our memory block and repeatedly emit the minimum item to the output block (using priority queue). When output is full, write to the disk and empty it to keep collecting more items. When the head block of an input list is empty, read the next one. Runtime is the number of blocks of input data.

There are only $O\left(\log_{M/B}\frac{N}{B}\right)$ merge phases, not $O\left(\log_2\frac{N}{B}\right)$. So, runtime improves to $O\left(\frac{N}{B}\log_{M/B}\frac{N}{B}\right)$.

Can we do better? No! This is information theoretically optimal.

There is some sort of a mismatch here. Usually, sorting takes $O(N\log N)$ and searching takes $O(\log N)$. However, in external memory algorithms, sorting takes $O\left(\frac{N}{B}\log_{M/B}\frac{N}{B}\right)$ time while searching takes $O\left(\log_B N\right)$, even when we should be able to search in $O\left(\frac{1}{B\log\frac{M}{B}}\right)$. This is nonsense though because runtime should be larger than $O\left(\frac{1}{B}\right)$. However, people come up with "buffer trees' which generalize B-trees to batch inserts at cost of $\frac{1}{B\log\frac{M}{B}}$ per insert with latency.

In external memory algorithms, we need to know B. There is also a hierarchy on memory with more than just two layers (memory and hard disk). Charles Leiserson's group introduced the notion of "cache-oblivious algorithms" which are designed for external memory that achieves optimal external memory bounds without knowing B or M. How could this possibly happen?

Consider the matrix multiplication problem. Charles said multiplying $n \times n$ matrices is equivalent to multiplying a lot of $\frac{n}{2} \times \frac{n}{2}$ matrices, so we can do this recursively. When we go deep enough, we reach a point where matrices fit into the memory so we do not have to access the external memory anymore. So, we only have to analyze how long it takes to get down there.

15 Parallel Algorithms

There are two models for analyzing parallel algorithms. One model is the circuits of logic gates. The other model is parallel processors over shared memory. These models turn out to be equivalent from the theoretical perspective.

For the second model, the metric we can use to measure parallel algorithms' performance is the speed: the number of parallel steps to solve. Another metric is the number of processors used.

For the first model, the circuits of logic gates, we can measure the performance by the size (the number of the gates) or the depth (the layers of the circuit).

§15.1 Circuits

Consider the logic gates and/or/not (others are just combinations). The logic gates are "clocked," meaning the output is available one cycle after the input, so the depth determines the runtime.

"Fan-in" is the number of inputs to one gate. We can talk about bounded fan-in where each gate has a constant number of inputs, or unbounded fan-in where each gate has arbitrary number of inputs. For example, OR gate can have a bounded fan-in of 2. An OR gate with k inputs can be reduced to $k-1$ OR gates linked together as a balanced binary tree, requiring $\lg k$ layers.

Definition 15.1.1. $AC(k)$ is the set of all functions computable with a circuit of depth $O\left(\log^k n\right)$ using $\operatorname{poly}(n)$ gates and unbounded fan-in.

Definition 15.1.2. $NC(k)$ is similar to $AC(k)$ but with bounded fan-in. (NC = Nick's class. SC = Steve's class.)

Theorem 15.1.3

$$AC(k) \subseteq NC(k+1) \subseteq AC(k+1)$$

Proof. $AC(k) \subseteq NC(k+1)$: Replace each unbounded gate with balanced tree. $NC(k+1) \subseteq AC(k+1)$: Unbounded is always better than bounded. □

Definition 15.1.4. $AC = NC = \bigcup_k NC(k)$

§15.1.1 Addition

Consider the problem of adding two n-bit numbers. The grade school algorithm simply adds two bits starting from the lower order to higher order using an adder that accepts input of two bits and a carry and outputs result and carry. This algorithm uses depth $O(n)$ and size $O(n)$. (This is called "ripple adder.")

Another idea called "carry lookahead" has depth $O(\log n)$. Note that the key problem is figuring out the carry bits. If we know the carry bits, we can add each bit in parallel in constant time. (We are going to ignore the result bits now.)

The algorithm will pre-plan for hypothetical values of carry bits. Given inputs a_i, b_i and unknown c_{i-1} (carry from previous bit), what are the cases for the output c_i (this carry bit)? If $a_i = b_i = 0$, $c_i = 0$ regardless of c_{i-1} ("kill"). If $a_i = b_i = 1$, $c_i = 1$ ("generate"). If $a_i \neq b_i$, $c_i = c_{i-1}$ ("'propagate"). We will consider these three cases with a_i, b_i as parameters as gates. Given a carry bit, the final carry bit is the result of going through all these gates.

Define "multiplication table" for these operations composed together. See notes. Note that the operation is associative.

Note that the first gate is k (and more importantly, not p). By induction, $x_i \cdot x_{i-1} \cdot \ldots x_0$ is never p. So, this means we can get the value of c_i depending on whether the composition is k or g.

To summarize, given $x_i \in \{p, g, k\}$, for each c_i we need $x_i \cdot x_{i-1} \cdot \ldots \cdot x_0$. This is like prefix sum: $y_0 = x_0$, $y_i = x_i \cdot y_{i-1}$. This is like a ripple adder.

How could we compute y in parallel? Even simpler, how to compute y_n. Because of the associative property, we can simply write the parentheses so we compute this as a binary tree. We can naively compute all n prefix values by using n trees. Size would be $O(n^2)$ and depth is $O(\log n)$. However, note that there are a lot of overlaps. One subtree can be used for multiple outputs. Just like in segment tree, we can assemble any prefix from $\log n$ subtrees of the big tree. However, we cannot simply use high fan-out. We can try to feed subtree values up and then down instead, giving $O(\log n)$ depth and $O(n)$ gates.

"As a theoretical computer scientist, I am opposed on principle to thinking about constant factors." - Karger

§15.2 Parallel Random Access Machine (PRAM)

We have processors running synchronously. Each processor can perform local computation. They can also read or write a shared memory (RAM). We will have to worry about conflict resolution: What happens when two processors read/write the same location? If the conflict is reading, they all get the same value but slowly.

There is the "exclusive read" model which allows only one processor to read one memory location in one time step (similar to bounded fan-out). There is also "concurrent read" which allows many processors to read the same location (similar to unbounded fan-out).

Similar models exist for writing. We need to resolve issues with writing multiple values at once. Ideas include choosing arbitrary result, random result, max/min result, garbage, etc.

Acronyms include CRCW (concurrent read/write), CREW (concurrent read, exclusive write), EREW.

Definition 15.2.1. $CRCW(k)$ = problems solvable in $O\left(\log^k n\right)$ time using poly(n) processors CRCW.

Like in previous model, $CRCW(k) \subseteq EREW(k+1)$. $NC = \bigcup NC(k) = \bigcup CRCW(k) = \bigcup AC(k)$. Any circuit of depth d size g can be simulated by a PRAM with g processors in time d. Any PRAM algorithm with n processors and time t can be built into a circuit with $n \cdot \text{polylog}(n)$ gates and depth $t \cdot \text{polylog}(n)$.

§15.2.1 Finding OR

Note that there are meaningful differences between EREW and CRCW. Suppose we want to compute the OR of n input bits. Runtime for EREW is $O(\log n)$ using OR tree. For CRCW, except the garbage rule, we can only be sure that the written value is one of our desired written values. So, we should add a condition on the processor such that they only write 1 if they see a 1 and write nothing if they see a 0.

§15.2.2 Max of n values

For EREW, runtime is $\Theta(\log n)$, solved by doing tree comparison. For CRCW, runtime is $O(1)$ using n^2 processors to compare every pair. The max is the value which wins all $n-1$ comparisons. Each number checks if it is the max by computing the OR of "do I lose to some other number?" which we can already solve in $O(1)$. If so, write itself to the output.

§15.3 Work of an algorithm

Work of an algorithm is the number of processors times the time spent. In practice we only have a few processors, so one processor has to simulate multiple processors as demanded by our algorithm.

A parallel algorithm is called "efficient" if the amount of work is not more than the order of the best sequential runtime. (Parallelism often costs efficiency.)

§15.3.1 Max problem

For the max problem, we can solve in $O(\log\log n)$ time using n processors. Work is $O(n\log\log n)$ rather than $O(n^2)$. Suppose we have k candidate maxes remaining. Make $\frac{k^2}{n}$ groups of $\frac{n}{k}$ items, totaling to $\left(\frac{k^2}{n}\right)\left(\frac{n}{k}\right) = k$ items. Now we solve max per group in $O(1)$ time using quadratic-processor max algorithm. We will need $\left(\frac{n}{k}\right)^2\left(\frac{k^2}{n}\right) = n$ processors to reduce n items to $\frac{k^2}{n}$ items in $O(1)$ time.

Using $2n$ processors, runtime is:

$$T(k) = O(1) + T\left(\frac{k^2}{n}\right)$$

Write $k = \frac{n}{2^i}$. Then,

$$\begin{aligned} T\left(\frac{n}{2^i}\right) &= O(1) + T\left(\frac{1}{n}\frac{n^2}{2^{2i}}\right) \\ &= O(1) + T\left(\frac{n}{2^{2i}}\right) \end{aligned}$$

The denominator is squared in one step. After $\log\log n$ squares, the denominator is n so we are done.

§15.3.2 Parallel prefix sum

We can transform our circuit algorithm into PRAM algorithm, using $O(\log n)$ time on n processors. This is not work-optimal because the work is $O(n\log n)$ but the sequential algorithm runs in $O(n)$ time.

Can we improve? Can we solve this using $\frac{n}{\log n}$ processors? Try making blocks of $\log n$ contiguous x_i. Assign one processor per block. Have that processor compute $y_j =$ the product of all the j-th processor's items. Do parallel prefix on y_i using the old algorithm then have each processor fills in the prefix of its block starting from previous y_j prefixes.

Each processor does $\log n$ time on its block. Parallel prefix takes $\log n$ time on $\frac{n}{\log n}$ items. Total work is $O(n)$.

The appeal of work-efficient algorithm is that we can run this algorithm on machines with any number of processors. If $k < \frac{n}{\log n}$ then time is $\frac{n}{k}$, giving us a perfect speed up.

§15.3.3 List ranking

We are given a linked list of n items. The list is initially stored in an array of list nodes but with out-of-order pointers. For each item, find its "position" in the list: the number of items following it when the linked list is transformed into an array. This basically allows us to "unroll" the linked list into an ordered array.

Being able to solve this allows us to solve plenty of problems. For example, we can solve the parallel prefix problem but on this jumbled up list.

There is a trick called "pointer jumping." Let $d(x)$ be the value in each node and the next pointer $n(x)$. Pointer jumping will do $d(x) += d(n(x))$ then $n(x) = n(n(x))$. Doing this $O(\log n)$ time gives us all prefix sums. If we initially set $d(x) = 1$ for all x we have the list ranking.

The work is $O(n \log n)$. We can make this work-efficient by using randomization. Then, we can derandomize to $O(\log n \log \log n)$ time using $\frac{n}{\log n}$ processes.

§15.4 Parallel sorting

Can we do parallel binary search on CRCW? Yes. With k processors, runtime is $O(\log_k n) \subset O\left(\frac{\log n}{k}\right)$. On each iteration, we divide the array up into k blocks. Each processor considers the first item of each block and answers whether the query is to the left or to the right of that item. Then, each processor compares with its neighbor. For the two processors that point toward the middle, they can answer that the query should be somewhere there. So, we have gone down from n items to $\frac{n}{k}$ items. Repeat this until we reach one item.

For sorting, note that we can find max and remove items in $O(1)$ time. So, we have $O(n)$ sorting algorithm using n^2 processors which is pretty bad.

Idea: for each element, compare to all others and count the number of smaller items. That gives us the index. Put each item at that index. We use n^2 processor for comparison. Counting smaller items uses n^2 processors and $O(\log n)$ time (using the aggregation tree/pointer jumping method). Putting each item at that index uses n processor with $O(1)$ time. So, overall we can sort in $O(\log n)$ time using n^2 processors. The work is $O(n^2 \log n)$ though, which is not efficient.

Cole figured out a way to parallelize merge sort. Merge sort has $O(\log n)$ phases of merging. If you can make merging fast using parallel computations, then we have a fast sorting algorithm.

Idea: each item x in list A finds its position in list B (and similarly for items in B). This yields the total number of items preceding x. So, we can order them to right index. Each item uses one processor for binary searching. If we have two length k lists, then we need $2k$ processors running in $O(\log k)$ time.

In merge sort, total size of list is n in each phase. We need n processors overall. Time is $\log k \leq \log n$. Each of our merge phase runs in $O(\log n)$ time using n processors. Overall, sorting is done in $O(\log^2 n)$ time using n processors. Work is $O(n \log^2 n)$.