CS 670: Advanced Analysis of Algorithms

David Kempe

Fall 2022

Welcome to CS 670: Advanced Analysis of Algorithms. Here's some important information:

- The course webpage is http://david-kempe.com/CS670/index.html
- David's office hours are 1-2pm in SAL 232 on Mondays, and Yusuf – the TA – has office hours from 10-11am on Thursdays in SAL 246.
- The final exam is scheduled for 8-10am on Monday, December 12, but it may be changed to a take-home final later in the semester.
- These notes were taken by Julian and are sure to contain some typos and omissions, due only to Julian.

Contents

§1 Monday, August 22

Let's start with a high-level question: what is computer science? Why isn't computer science the same thing as algorithms? After all, anytime anyone runs something on a computer, they have used (or even created!) an algorithm. Why don't we call the department the Department of Algorithms?

The primary distinction is that when doing algorithms work, we value different things than other fields of computer science (that also utilize algorithms). Namely, we care about:

- Provable guarantees (on time, space, performance, correctness, etc.),
- Generality,
- Simplicity,
- Mathematical insight underlying an algorithm.

Consequently, this class will involve lots of:

- Proofs,
- Abstraction,
- Mathematical modeling,
- Formal thinking.

§1.1 Shortest paths

Let's work on the problem of finding shortest paths with non-negative edge weights. First, let's formalize this slightly.

Problem 1.1 (Shortest paths)**.** Given a directed graph $G = (V, E)$ with edge costs $c_e \geq 0$, a source $s \in V$ and sink $t \in V$, find a shortest path (i.e., path with minimal sum of edge weights) from s to t.

Example 1.2

Why are we solving this problem? What are some possible applications? There's:

- (driving) directions,
- routing in a computer network,
- routing physical packets,
- distance in (social) networks,
- puzzles.

Let's go over the familiar solution to this problem.

Algorithm 1.3 (Dijkstra) —

1. Start with $S = \{s\}$
2. Set $d[s] = 0, d[v] = \infty \ \forall v \neq s$
3. While $t \notin S$:

   a) Find node $v \notin S$ minimizing $\min_{u \in S} d[u] + c_{(u,v)}$
   b) Add v to S and set $d[v] = \min_{u \in S} d[u] + c_{(u,v)}$
4. Return $d[v]$

Next time we'll discuss correctness of this algorithm — what exactly do we even mean when we say that the algorithm is correct? — and we'll get into minimum spanning trees.

§2 Wednesday, August 24

§2.1 Dijkstra's algorithm

Last time we were discussing **Dijkstra's algorithm** for finding (lengths of) shortest paths in directed graphs with non-negative edge weights.

Algorithm 2.1 (Dijkstra) —

1. Start with $S = \{s\}$
2. Set $d[s] = 0, d[v] = \infty \ \forall v \neq s$
3. While $t \notin S$:
   a) Find a node $v \notin S$ minimizing $\min_{u \in S} d[u] + c_{(u,v)}$
   b) Add v to S and set $d[v] = \min_{u \in S} d[u] + c_{(u,v)}$
4. Return $d[v]$

Note that we can keep track of parent nodes – i.e., set `p[v] = u` – if we'd like to compute the actual shortest path from s to t, rather than merely its length. Now it's time to prove correctness of our algorithm.

Proposition 2.2
At the conclusion of Dijkstra's algorithm, $d[t] = \text{dist}(s,t) := \min_{\text{path p}} \sum_{e \in p} c_e$.

Proof. We claim further that for all $v \in S$, $d[v] = \text{dist}(s,v)$. We induct on $|S|$. When $|S| = 1$, we have $d[s] = 0$ which indeed equals $\text{dist}(s,s)$, as our graph has no negative edges.

Now suppose we are adding a new v to S. First note that we do not change the values $d[u]$ for all u previously in S. Now we show $d[v] = \text{dist}(s,v)$.

(a) Note that $d[v] \geq \text{dist}(s,v)$, as

$$d[v] = c_{(u,v)} + d[u] = c_{(u,v)} + \text{dist}(s,u)$$

is the length of *some* path from s to v.

(b) To see $d[v] \leq \text{dist}(s,v)$, suppose otherwise, namely that $d[v] > \text{dist}(s,v)$. Then there is a strictly shorter $s \to v$ path p. At some point p crosses from S to $\overline{S}$, via an edge (u', v').

Because edge costs are nonnegative, $\text{dist}(s,v') \leq \text{length}(p) < d[v]$. By our inductive hypothesis, $d[u'] = \text{dist}(s,u')$, so $d[u'] + c_{(u',v')} = \text{dist}(s,v') < \text{dist}(s,v) < d[v]$. This produces contradiction with the definition of v.

So $\text{dist}(s,v) \le d[v] \le \text{dist}(s,v)$, and we're done. □

Now let's think about the time complexity of this algorithm. With a naive implementation, we get $\Theta(N \cdot M)$, for N the number of nodes and M the number of edges. In particular, the loop in (3.) occurs at most N times and line (3a.) has cost M, as it checks all the edges on vertices in S.

To make things even more efficient, we'll want a data structure that efficiently supports the ranking of our nodes by their estimated distance from s. In particular, we'll want a data container with the following operations:

- insert,
- decrease value,
- find min,
- remove.

With a standard min heap, we get complexities of $O(\log n)$ for all operations, with $O(1)$ for find-min. Furthermore, we call insert, find-min, and remove N many times, while we call decrease value M times. So this implementation puts Dijkstra in $\Theta((M+N)\log N)$.

Usually $M > N$, so this is dominated by $M \log N$ and we'd really like to make decrease value a constant-time operation if possible. This would make our implementation $O(N \log N)$, and indeed we'll see how to do that — at the level of amortized analysis — using Fibonacci heaps later on.

§2.2 Minimal spanning trees

Problem 2.3 (Minimal Spanning Tree)**.** Given an undirected connected graph $G = (V, E)$ with edge costs $c_e \ge 0$, find a set T of edges with minimal total cost so that (V, T) is connected.

Remark 2.4. There's an easy pre-processing step if edge costs are negative, where you just grab all the negatively weighted edges, glue the connected components into single vertices, and restart with a problem formulated in the sense of Problem 2.3.

Algorithm 2.5 (Kruskal's algorithm) — Go through the edges E sorted from cheapest to most expensive, adding each to T if they don't create a cycle.

Algorithm 2.6 (Prim's algorithm) — Start with $T = \varnothing$ and $S = \{s\}$ for arbitrarily chosen $s \in V$. While $S \neq V$, let $e = (u, v)$ be the cheapest edge connecting S to $\overline{S}$, and add e to T and v to S.

Lemma 2.7

Kruskal's and Prim's algorithms output spanning trees.

Proof. First we show that the outputs are acyclic. For Kruskal's this is immediate and for Prim's it is because each added edge e produces a new leaf.

To see that the outputs are connected, suppose otherwise for Kruskal and consider its output (V, T). Because G is connected, there are edges in G leaving a connected

component S of (V,T). Then at some point we considered a cheapest edge $e \in E$ from S to $\overline{S}$ and skipped it, producing contradiction.

For Prim's, first note that the algorithm terminates because G is connected. Furthermore, it only terminates when $S = V$, meaning (V,T) is connected. □

Lemma 2.8

Kruskal's and Prim's algorithms produce *minimal* spanning trees.

Before we prove this, we'll want something called the **cut property**.

Lemma 2.9

Fix a graph $G = (V,E)$ with distinct edge costs $c_e \geq 0$. If $e \in E$ is the cheapest edge for any cut $(S,\overline{S})$ of G, then e is part of the (unique) minimum spanning tree for G.

Proof. Let T be an MST, and assume $e \notin T$. Then $T \cup \{e\}$ has a cycle C containing an edge $e' \in T$ that crosses $(S,\overline{S})$. So we can replace e' with e in T and get a spanning tree of strictly lower cost, producing contradiction. □

Now we can return to Lemma 2.8, proving it by showing that every $e \in T$ chosen by Prim/Kruskal is the cheapest across some cut. Then we'll have that T is a subset of the MST on G and thus that T indeed equals the MST.

Proof of Lemma 2.8. For Prim's algorithm, note that any $e \in T$ is cheapest across $(S,\overline{S})$ when added. For Kruskal, note that when e is added, it connects two current connected components S, S', and thus it is cheapest for $(S,\overline{S})$ (or $(S',\overline{S'})$, if you like). □

§2.3 Matroids

It's pretty remarkable when greedy algorithms turn out to be optimal, as they make fundamentally local moves yet somehow end up with (globally!) optimal solutions. In the case of the MST problem, the idea is that picking up a cheapest edge e does not force your hand further down along the line. That is, picking e does not force you to pick edges e' or e'' later on as you construct your tree; you will instead still have freedom to choose.

Formalizing this property gives rise to the notion of matroids, for which it can be shown that Kruskal's algorithm is always correct and optimal!

Definition 2.10 — A **matroid** is a set system $(X,\mathcal{I})$ with X a set and $\mathcal{I} \subseteq \mathcal{P}(X)$ a collection of *independent sets* with the following properties:

1. $\varnothing \in \mathcal{I}$,
2. $S \in \mathcal{I} \implies \mathcal{P}(S) \subseteq \mathcal{I}$, (downward closed)
3. $S, S' \in \mathcal{I}$ and $|S| < |S'| \implies \exists x \in S' \setminus S$ s.t. $S \cup \{x\} \in \mathcal{I}$. (exchange property)

Example 2.11 1. For any set X, $(X,\mathcal{P}(X))$ is a matroid.

2. For any set X and $k \in \mathbb{N}$, $(X, \{S \subseteq X \mid |S| \leq k\})$ is a matroid, known as a *uniform matroid*.

3. For any vector space X and collection $\mathcal{I}$ of linearly independent subsets of X, $(X, \mathcal{I})$ is a matroid.

§3 Monday, August 29

§3.1 Matroids II

Last time we were talking about matroids. One natural question is why do we call the $\mathcal{I}$ in a matroid $(X, \mathcal{I})$ the *independent sets* of the matroid? The reason why is precisely because of the third example we looked at last time, namely the fact that that $(X, \mathcal{I})$ is always a matroid for X a subset of a vector space and $\mathcal{I}$ the collection of all linearly independent subsets of X. This is known as a **linear matroid**. Another example is to let X be the edges of a graph G and $\mathcal{I}$ be the set of all forests of G (i.e., acyclic edge sets). This one's known as a **graphical matroid**.

Proposition 3.1

If $(X, \mathcal{I})$ is a matroid and each $x \in X$ has a weight w_x, then running Kruskal – modified to pick the largest element – will find a max-weight independent set $I \in \mathcal{I}$.

Remark 3.2. When we say *maximum* we mean largest, and when we say *maximal* we mean that nothing can be added. So being maximal is a local condition while being maximum is a global condition.

§3.2 Binomial heaps

Recall that Prim and Dijkstra's algorithms each need data structures supporting `insert`, `find`/`delete`, and `decrement`.

	Heap	Fibonacci Heap	# calls
`insert`	$\Theta(\log N)$	$\Theta(1)$	N
`find/delete_min`	$\Theta(1)$ / $\Theta(\log N)$	$\Theta(\log N)$	N
`decrement`	$\Theta(\log N)$	$\Theta(1)$	M

To understand the efficiency of the Fibonacci heap, we'll use **amortized analysis**, which understands the worst-case cost of a *sequence* of operations rather than a single operation. A crux idea is to pretend that some operations take longer than they truly do and to use that to pay for (more expensive) future operations.

Before we get to Fibonacci heaps, though, we'll start things off with binomial heaps.

Definition 3.3 — The **binomial trees** $(B_i)_{i \in \mathbb{N}}$ are defined by B_0 being an isolated node and B_k being a node with k children, namely $B_0, \ldots, B_{k-1}$.

Alternatively, one can define B_k as two copies of B_{k-1} with a node between their roots, one of which is arbitrarily determined to be the root of the combined tree. This makes it easy to see that B_k has 2^k many nodes. We say k denotes the **rank** of the binomial tree.

Definition 3.4 — A **binomial heap** is a finite collection of binomial trees, each satisfying the heap property (i.e., the minimal element of each subtree is its root),

such that there is at most one tree of each rank.[a]

[a] We'll violate the 'at most one tree of each rank' condition later on.

Remark 3.5. For concreteness, we can assume that a binomial heap h is implemented as an array of the roots of the binomial trees constituting h. We index these trees (or roots of trees) by rank, as no two trees in a given heap are permitted to share the same rank.

The operations we'd like our binomial heap h to enjoy are:

- `insert(x, h)`: inserts a new element x into h,
- `meld(h, h')`: combines binomial heaps h, h' into one new binomial heap,
- `delete-min(h)`: deletes the minimum from h.

For `meld()`, an issue really only arises if we have binomial trees in `h` and `h'` of the same rank. In this case, going from smallest to largest rank, if we have two trees of rank k, we can combine them into one of rank $k+1$ and repeat.[1] (This is like binary addition!) It should be easy to see that this terminates.

For `insert()`, we can create a new $B_0 = \{x\}$ and meld with `h`. For `delete-min()`, we can check all the roots to find the minimum and delete it. Then the children form a new heap `h'`, which we can meld with the rest of `h`.

§4 Wednesday, August 31

§4.1 Binomial heaps II

Last time we talked about the implementations of `insert()`, `meld()`, and `delete-min()`. To think about their complexities, note that a binomial heap with n elements has maximum rank at most $\log_2 n$ and thus at most $O(\log n)$ trees & roots. An immediate consequence of this observation is that all our operations are in $O(\log n)$, since `meld(h, h')` is linear in the the number of roots of its arguments and `insert(x, h)` and `delete-min(h)` are just `meld()` with at most $O(\log n)$ extra steps.

Let's see if we can be even more refined with this analysis, though, to see that `insert`s are cheaper than we think – at least at the level of amortized analysis. Informally, the idea is that adding 1 to an n-bit number rarely takes a full n many steps of work. (For instance, if n is even then it takes only 1 step.)

Simply put, expensive insertions seem to be rare, so we want to amortize the analysis to get $O(1)$ insertion cost (amortized). The credit invariant we want is that each binomial tree's root holds on to credit that can pay for a future `merge` with another tree. We'll set this up like so:

- `insert`: when creating a new B_0 for `x`, we do one extra unit of work to give B_0 a credit.
- `delete-min`: for rank k, spend an extra k units of time to give each new tree a credit. This makes `delete-min` a constant 2 times more expensive.
- `meld`: when we merge two trees into one, one credit pays for the merge and the other stays with the new tree.

[1] We set the root of the combined tree to be the smaller of the two original roots.

So now `meld()` and `delete-min()` are amortized $O(\log n)$ and `insert()` is amortized $O(1)$.[2] But we think we can do *even better*! To make `meld()` amortized $O(1)$, we can change it to a NO-OP.[3] Then we're allowing for multiple trees of the same rank at the same time. Any time we run `delete-min`, we can first do a full cleanup pass merging all duplicates, which costs $O(\log n)$ with merges paid by credits.

Now how should we implement `decrement`? A first attempt is that after locating the right node v, we can decrease its value and swap up as in regular heaps. This is $\Theta(\log n)$, though, and we can't amortize well because you can build inputs where a large fraction of decrements need this many swaps. We really want amortized $O(1)$.

§4.2 Fibonacci heaps

An alternative solution is to remove v and its subtree. This restores the heap property because the parent is unaffected and because v has been made smaller, so its children are still larger than it. A new problem is that we don't end up with binomial trees anymore! But why did we need binomial trees? The only crucial property was that they ensured that the number of trees be logarithmic in the total number of nodes.

We can preserve this property while making the overall picture more flexible in the following away: allow for removals of subtrees for `decrement`, but as soon as a node v loses a second child, it (and its remaining subtree) must be removed. Note that this could propagate to the parent of v and further up. This is known as the **Fibonacci heap rule**.

The goal now is to show that – with the Fibonacci heap rule – the number of nodes in a tree is still exponential in the rank of the tree. After cleanup, we have at most one tree for each rank, implying that there are at most $O(\log n)$ trees. Then `delete-min()` just needs to check $O(\log n)$ roots, as desired. The only real obstruction to this goal is the possibility that one removal leads to a long chain of removing ancestors, giving rise to super-constant time. But for this to happen, all these ancestors must already have lost a child earlier, which was the first child (and thus a cheap removal). So we can amortize.

Formally, we can add *removal credits* to the picture: when v loses its first child, that removal does constant extra work to give v a removal credit. When v loses its second child, the removal credit pays for removing r and its remaining subtree. In addition, we'll need to add a merge credit for v and its parent when v loses its first child.

With this, we have that `decrement()` and `insert()` are amortized $O(1)$, and `delete-min()` is amortized $O(m)$ for m the max rank tree we have. To work out the max rank with n nodes, let's lower-bound the minimum number of nodes of a tree of rank k, call it s_k. Then you can see that $s_k = s_0 + \sum_{i=0}^{k-2} s_k$. So $s_k = s_{k-1} + s_{k-2}$, explaining where "Fibonacci" comes in. Then $s_k \approx \phi^k$ for ϕ the golden ratio, and we're done :)

§5 Wednesday, September 7

Today we'll be talking about data structures that support Kruskal's algorithm, which should be even more appealing now that we know — from Proposition 3.1 — that Kruskal's works for arbitrary matroids (not just MSTs!).

[2] It's not so trivial to me that `insert()` is amortized $O(1)$ while `meld()` is still amortized $O(\log n)$, but David gave a helpful explanation that I'm not sure I can recreate.

[3] This means 'do nothing', apparently.

§5.1 Implementing Kruskal

So how do we implement Kruskal's algorithm? The naive implementation is, upon reaching an edge that you'd like to check does not create a cycle in your set of selected edges S, to run BFS between its endpoints (among edges in S). The complexity of this is $\Theta(M \cdot N)$, since we're running M many BFS's, each of them on a set of at most $N-1$ edges.

An alternative is to maintain an array that gives the index of the connected component for each node. Initially, each node is its own component, and when two components merge via an edge, we overwrite one with the other. This still takes linear time for each merge operation, since we need to find all nodes belonging to a particular component.

§5.1.1 Union-Find

To avoid linear overwriting, we can instead have pointers to 'parent' components. Initially, each node points to itself. When two components merge, we change the root pointer of one to point to the other. This leads us to the **Union-Find** data type, which supports a collection of disjoint sets with the following two operations:

- `Find(e)`: returns the 'index' of the set that `e` belongs to,
- `Union(e1, e2)`: takes the union of the sets containing `e1` and `e2` in our collection.

Our implementation of the data type is as a directed forest. The root of a tree is its 'representative.'

- `Find(e)`: follow pointers from `e` to its parent until reaching the root,
- `Union(e1, e2)`: `Find(e1)` and `Find(e2)`, then make one the parent of the other.

If we allow for any choice in implementing `Union()`, we can end up with trees that are just chains, giving rise to linear-time `Find()`. This is pretty easy to fix: we should make smaller trees point to larger trees. For the notion of "largeness", we can use either height or number of nodes; it turns out that they both work. So when two sets merge, we'll point the root of a smaller one to a larger one.

To analyze this, note that the height above a node can only increase when the size of the node's set at least doubles. So max weight is $\leq \log_2 n$, and all operations run in $O(\log n)$. Now Kruskal demands $O(m \log m)$ for sorting followed by $O(m \log n)$, so it's in $O(m \log m)$.

Remark 5.1. There's a somewhat related idea of **path compression** for making root look-up fast in trees. The idea is that whenever we search for the root from a node, we do a second pass over that path and point everyone directly at the root. This doesn't really help worst-case running time, but it gets amortized time down to $O(\log^* n)$.

§5.2 Dynamic programming

> *The first thing to know about dynamic programming is that there's nothing dynamic about it and it has nothing to do with programming.* – David

Dynamic programming can be a powerful tool to help solve problems that seem truly intractable at first glance. Let's begin with an example. Say you have a triangle of numbers, and you'd like to take a path from the root to a leaf while accumulating the largest sum possible. An exponential-time solution is to try all paths, e.g., with a

recursive approach that tries the left and right subtriangles and then adds the root to the larger of the two. This gives time about 2^{n-1} for n rows.

Note that this is very wasteful, though: solutions to the same subproblems are frequently recomputed. They should really be stored/**memoized** instead. Even better, we should run this bottom up, computing the lowest level first, then the level above it (using the stored values), and so on. With this idea, you get an $\Theta(n^2)$ algorithm!

Abstracting a bit, the key property here was *optimality of subproblems*, or the **principle of optimality**. That is, the optimal solution to a subproblem can be used directly in the optimal solution for the bigger problem. This is not always the case; imagine this triangle problem but in which you're not allowed to repeat the same number on your way down to a leaf. That's a pretty reasonable problem, but it throws our dynamic programming solution out the window.

Example 5.2

Say we want to multiply matrices $A_1 \cdot A_2 \cdot \cdots \cdot A_k$, where $A_i \in \mathbb{R}^{n_i \times m_i}$. The objective is to achieve this with as few scalar multiplications as possible, making use of the fact that matrix multiplication is associative.

To be concrete for a moment, say we're multiplying matrices with shapes $(1 \times n)$, $(n \times 1), (1 \times n)$. Then if we start by multiplying the left two, we do only $2n$ work, whereas if we start by multiplying the right two, we do $2n^2$ work. That's a big difference!

Returning to the original problem, a key observation is that there is some *final* multiplication we perform, of the form $(A_1 \cdot \cdots \cdot A_i) \cdot (A_{i+1} \cdot \cdots \cdot A_k)$. Before we get to that multiplication, we need to compute both of the constituent terms optimally. This gives us a key ingredient of the dynamic programming approach! Write $OPT(i,j)$ to denote the optimum cost to compute $A_i A_{i+1} \cdots A_j$. Then we have

$$OPT(i,j) = \min_{i \le \ell < j} OPT(i,\ell) + OPT(\ell+1,j) + n_i \cdot m_\ell \cdot m_j.$$

(Along with the base case $OPT(i,i) = 0$.) If you were to code this up, the outer loop would be over $j - i$ and the inner loop over i.

§6 Monday, September 12

§6.1 Dynamic programming II

Let's look at another classic example of dynamic programming.

Example 6.1 (Knapsack problem)

Here's the problem: given n items with weights w_i and values v_i, as well as a total weight limit W, select a subset of items with maximum total value and total weight within W.

A natural choice to set up the DP approach is to have $OPT(i,w)$ be the best you can do with weight limit w on items $\{1,\dots,i\}$. Then there's an easy recurrence

$$OPT(i,w) = \max\big(OPT(i-1,w), v_i + OPT(i-1, w-w_i)\big).$$

This gives us an $\Theta(nW)$ solution[a], which seems pretty good! Surely that's a

polynomial-time solution, so we're done. But the knapsack problem is NP-complete, so what's going on here? The issue is that $\Theta(nW)$ is not really polynomial. We call an algorithm **polynomial time** when its running time is polynomial in the input size (i.e., the number of *bits* in the input).

Usually, this relates closely with natural measures of our input like array size, number of nodes/edges in a graph, etc. But not necessarily when the input involves numbers and we do more than arithmetic with the numbers. In particular, the number W takes $\Theta(\log W)$ bits to write, so W can be exponential in the input size.

[a] We're assuming weights are integral, which is fairly benign.

There's a relevant definition to describe this phenomenon!

Definition 6.2 — An algorithm has **pseudo-polynomial** running time if the running time would be polynomial if all numbers in the input were written in unary.

Informally, pseudo-polynomial time captures polynomial dependence on the values of our inputs, not their sizes.

Example 6.3 (Subset sum)

Problem: given n numbers $a_1, \ldots, a_n$ and target t, find a subset of the a_i adding up to t. Rather than doing more work, we can just reduce to Knapsack! We set $w_i = v_i = a_i$ for all i and $W = t$. Then we get a value of t in our knapsack if and only if a set summing to t exists.

§6.1.1 Independent set on trees

Definition 6.4 — For a graph $G = (V, E)$, a set $S \subseteq V$ is **independent** if no pair of edges in S is connected with an edge.

Problem 6.5 (Weighted independent set)**.** Given a graph G with weighted edges, select an independent set S of maximum total weight.

This problem is NP-hard, and furthermore hard even to approximate. For the moment, we'll look at the special case where G is a tree. Then, to do dynamic programming, "natural" subproblems are subtrees. We'll want to think of trees as rooted, so let's start by picking an arbitrary root of G. We'll then write $OPT(v, \text{“}Yes\text{”}/\text{“}No\text{”})$ for the maximum weight independent set in a subtree rooted at v which may / may not include v itself. Our base case will be $OPT(\perp, \text{“}Yes\text{”}/\text{“}No\text{”}) = 0$, where $\perp$ is the empty tree (i.e., dummy leaf).

Remark 6.6. The general approach for DP on trees is exhaustive search over the relatively few choices at the root, which are then passed to its children as parameters for their optimal solution.

§7 Wednesday, September 14

§7.1 Shortest paths (with negative edge weights)

Problem 7.1. Given a directed graph $G = (V, E)$ with (possibly negative) edge costs $c(e)$ and $s, t \in V$, find a path p from s to t minimizing $\sum_{e \in p} c(e)$.

Remark 7.2. This problem is totally ill-posed, as we're allowing G to have negative cycles. For now, we'll define this problem away and assume G contains no negative cycles. Later, we'll see how to actually test that.

Recall that Dijkstra generally does not work with negative edge weights. So let's try to crack this with a dynamic programming approach. A useful subproblem is to consider shortest paths from all nodes v to t. So we'll define $OPT(v)$ to be the the minimum total cost of a path $v \to t$. Then $OPT(t) = 0$, because we have no negative cycles. And for $v \neq t$,

$$OPT(v) = \min_{(v,v') \in E} c((v,v')) + OPT(v').$$

Once again, we need to think about the principle of optimality (or *optimal substructure*) to ensure that this equality holds, though in this case it's pretty clear. But how would we actually implement this recurrence (in code)? In what order would we iterate over all $v \in V$? It's really not clear. We cannot easily determine an update order (or, relatedly, an induction variable for the correctness proof). To sidestep this problem, we introduce the number of hops as a second variable for the optimum solutions to subproblems.

Now we have

$$OPT(v,i) = \min\Big(OPT(v,i-1), \min_{(v,v') \in E} c((v,v')) + OPT(v',i-1)\Big).$$

And our base case is $OPT(v,0) = 0$ if $v = t$ and ∞ otherwise. Now this implementation can actually be turned nicely into a bottom-up solution. Furthermore, we only need to compute up until $i = n-1$, as cycles aren't helpful. And we're done! Our code would look something like:

```
for all v: a[v][0] = \infty;
a[t][0] = 0;
for (i=1; i \leq n-1; ++i)
    for all v:
        a[v][i] = min(a[v][i-1], min_{(v, u)} c[v][u] + a[u][i-1])
return a[s][n-1]
```

Remark 7.3. We can reduce space usage in the previous implementation by replacing i and $(i-1)$ with their residues mod 2. In particular, we only ever need the previous row in our table.

In fact, it turns out that we can even furthermore save space by modifying this algorithm to be in-place on an array of size n. That's fairly remarkable! The implementation is as follows.

```
for all v: a[v] = \infty;
a[t] = 0;
for (i=1; i \leq n-1; ++i)
    for all v in any order:
        a[v] = min(a[v], min_{(v, u)} c[v][u] + a[u])
return a[s][n-1]
```

The proof of correctness follows from two observations. First, after i iterations, $a[v] \le OPT(v,i)$. Second, after i iterations, $a[v] \ge OPT(i,\infty)$. So after $n-1$ iterations, $a[v] = OPT(v,\infty)$! Furthermore, if in any iteration there's no update to any of the $a[v]$, then we can terminate the algorithm early.

Also, this algorithm — **Bellman-Ford** — naturally parallelizes! Each node repeatedly asks its neighbors for their current distance to t and then updates its own estimate. In fact, this idea underlies **distance vector protocols** for routing in networks. For each major destination (IP address or cluster thereof), routers maintain estimated distances (latency) as well as the cost of the first hop. And these are repeatedly updated.

In fact, rather than repeatedly asking out-neighbors for distances, it's more efficient to actively push new shorter distances to the in-neighbors when they arise. This is known as a **push protocol** rather than a **pull protocol** (i.e., pushing information to people that need it rather than asking information from people who may have it for you). One issue here is robustness; if routers go down, or even if some costs become more expensive, then you might need to do lots of recomputation. It's also vulnerable to 'impersonation', where some agents claim short distance to other destinations in order to attract traffic.

An alternative is **path vector protocol**, which keeps track of the entire path to a destination. A common one is BGP.

§7.1.1 Detecting negative cycles

So far, we've been relying on the assumption that G has no negative cycles. What if we didn't have this assumption and instead wanted to detect whether G has a negative cycle? One solution is to add a new vertex t and connect all nodes to it with edges of cost 0. Then we look for a negative cycle that can reach t. To do so, we check if any updates happen after round $n-1$ of Bellman-Ford.

Proposition 7.4

There is a negative cycle if and only if $OPT(v,n) \neq OPT(v,n-1)$ for at least one node v.

Proof. It only remains to show the forward direction. We'll use the contrapositive, i.e., we assume that $OPT(v,n) = OPT(v,n-1)\ \forall v$. Then $OPT(v,k) = OPT(v,n-1)$ for all $k \ge n-1$, because of our update rule. So $\lim_{k\to\infty} OPT(v,k) = OPT(v,n-1) > -\infty$, and there can't be negative cycles. □

§8 Monday, September 19

§8.1 Dynamic programming III

Let's wrap up dynamic programming today. Recall that last time we tried to solve the problem of shortest paths with negative edge weights (but nonnegative cycles). We started with a simple and correct recurrence:

$$OPT(v) = \min_{(v,v')\in E} c((v,v')) + OPT(v').$$

But we lacked a **computation order** to be able to turn this into a true computational solution. We remedied this by introducing the maximal number of hops k in a given path — which gave us our computation order — and then we tweaked things even further to reduce our solution's space complexity.

§8.1.1 Edit distance / sequence alignment

Problem 8.1. Given two strings x, y of lengths n, m, how "similar" are they?

There are some pretty natural candidates here:

1. Hamming distance (i.e., number of entries in which x and y differ), possibly weighted.
2. Longest (perhaps consecutive) common substring.
3. Number of "operations" needed to transform x to y.

In general, you may want to consider a natural model or "process" which probabilistically generates y from x. This may suggest particular measures, which detect the likelihood of y being noisily generated from the 'ground truth' x. That is, what is the assumed generative process that might cause these noisy observations? Based upon this, one can select a measure.

Today, we'll be considering candidate #3, which turns out to subsume #1 and #2 (without the requirement that the common substring be consecutive). First, we need to be precise about the operations we allow. They will be like so:

- Insert character into x: cost α
- Delete character from x: cost α
- Replace a character: cost β.

An alternative view is that of aligning x with y. So you take their characters in order, insert blanks in both, and you have a cost of α per blank and β per misaligned character.

Remark 8.2. Applications here include spell checking and DNA sequence alignment.

So let's attack this with dynamic programming. We'll define $OPT(i, j)$ to be the minimum cost for aligning $x[1, \ldots, i]$ with $y[1, \ldots, j]$. Then there are three options for $OPT(i, j)$:

- Match $x[i]$ with blank (and pay α).
- Match $y[j]$ with blank (and pay α).
- Match $x[i]$ with $y[j]$ (and pay β if they differ).

As we match the rest of the strings optimally, we're left with an obvious recurrence and with the base cases $OPT(i, 0) = OPT(0, i) = i \cdot \alpha$.

Remark 8.3. Dynamic programming can almost always be viewed as a shortest path computation in a suitable graph on subproblems. When maximizing, flip signs of all "values" to "negative costs." Table computation usually gives a DAG, meaning no cycles, and in particular no negative cycles. So most table computations are in retrospect optimized Bellman-Ford.

§8.2 Max-flow / min-cut

Problem 8.4 (Max-flow)**.** Fix a graph $G = (V, E)$ with source $s \in V$, sink $t \in V$, and a capacity for each edge $c_e \geq 0$. (For now we'll assume capacities are integral.) Now route as much "flow" as possible from s to t using edges.

Definition 8.5 — An **s-t flow** is a mapping $f : E \to \mathbb{R}^{\geq 0}$ with the following properties:

1. $f_e \geq 0 \ \forall e$ (nonnegativity),
2. $f_e \leq c_e \ \forall e$ (capacity),
3. $\sum_{(u,v)} f_{(u,v)} = \sum_{(v,u)} f_{(v,u)} \ \forall v \notin \{s,t\}$.

The **value** of a flow f, denoted $\Delta(f)$, is the total flow out of s (equivalently, the total flow into t).

An alternative definition is to give s-t paths $P_1, \ldots, P_k$ with flow values $\alpha_1, \ldots, \alpha_k \geq 0$ such that $\sum_{P_i \ni e} \alpha_i \leq c_e$. Obviously, the value of this flow is $\sum_i \alpha_i$.

Definition 8.6 — An **s-t cut** is a partition $(S, \overline{S})$ of V with $s \in S$, $t \in \overline{S}$. The *capacity* of $(S, \overline{S})$, denoted $c(S, \overline{S})$, is the sum of capacities of edges from S to $\overline{S}$.

Problem 8.7 (Min-cut)**.** Find an s-t cut minimizing $c(S, \overline{S})$.

Min-cut and max-flow seem well-motivated enough to begin with — as flowing through graphs is pretty common — but it furthermore turns out that lots of important problems reduce to min-cut / max-flow, making them even more worthy of our attention. In fact, it even turns out that they're equivalent!

Theorem 8.8 (Ford-Fulkerson)
Let $G = (V, E)$ be a graph with edge capacities $c_e \geq 0$ and $s, t \in V$ a source and sink. Then:

1. There is a poly-time algorithm computing a max s-t flow f^* and minimum s-t cut $(S^*, \overline{S^*})$;
2. Furthermore, $\Delta(f^*) = c(S^*, \overline{S^*})$;
3. If all c_e are integral, then all f_e^* are integral.

§8.2.1 Application: maximum bipartite matching

Problem 8.9 (Maximum bipartite matching)**.** Fix a bipartite graph $G = (V, E)$, i.e., with $V = X \sqcup Y$ and $E \subseteq X \times Y$. Find a maximum cardinality matching in G.

Recall that a **matching** is a subset of edges $E' \subseteq E$ such that no vertex is incident on more than one edge.

Then we can see that maximum bipartite matching reduces to max-flow as follows:

- Add a source s, connect to all $x \in X$ with capacity 1.
- Add a sink t, connect from all $y \in Y$ with capacity 1.
- Keep all edges $E \subseteq X \times Y$, and give them capacity $|Y|$ (though just 1 would work, and make the proof slightly harder).

§9 Wednesday, September 21

§9.1 Ford-Fulkerson

We'll spend most of today proving the famous Ford-Fulkerson theorem. Let's restate it first.

Theorem 9.1 (Ford-Fulkerson)
Let $G = (V, E)$ be a graph with edge capacities $c_e \geq 0$ and $s, t \in V$ a source and sink. Then:

1. There is a poly-time algorithm computing a max s-t flow f^* and minimum s-t cut $(S^*, \overline{S^*})$;
2. Furthermore, $\Delta(f^*) = c(S^*, \overline{S^*})$;
3. If all c_e are integral, then all f_e^* are integral.

Remark 9.2. Point 3 of Ford-Fulkerson *does not* imply that every max-flow has integral flow values when the capacities do, simply that *some* max-flow does.

Here's a high-level idea to implement the poly-time algorithm: while there is an s-t path P whose edges are not saturated, pick such a path, and add to it as much flow as possible with capacity constraints. Unfortunately, this turns out not to work at all: we might get stuck before we find a max flow.

The solution is to be able to undo flow on an edge, which can be regarded as sending flow on the edge in the opposite direction. Importantly, this idea makes use of the **residual graph** G_f with respect to a flow f, which is defined as follows:

1. For every e with $f_e < c_e$, it contains a "forward edge" with residual capacity $c_e - f_e$.
2. For every edge $e = (u, v)$ with $f_e > 0$, G_f contains the "backward edge" (v, u) with capacity f_e.

Algorithm 9.3 (Ford-Fulkerson) — Start with $f = 0$. While G_f contains an s-t path p, augment f along p.

Now we should define what it means to *augment* a flow f by a path p in a residual graph, but it really means what you think it means. Namely, let $\delta = \min_{e \in p} c'_e$. Then for all $e = (u, v) \in p$, increment f_e by δ if e is forward and decrement $f_{(v,u)}$ by δ if e is backward.

Now that we've specified our algorithm, we need to think about correctness and complexity. First things first: let's prove correctness and show that Ford-Fulkerson even outputs a valid flow. We proceed by induction on the iterations. When $f = 0$, we certainly have a valid flow. Now the inductive step:

Non-negativity: If $e = (u, v)$ is a backward edge, then $f_{\overline{e}}$ gets decremented by δ. But $f_{\overline{e}} \geq \delta$, since the minimum used to compute δ included c'_e.

Capacity: If e is forward, then f_e increases by δ, but the minimum includes $c'_e = c_e - f_e$, so $\delta \leq c_e - f_e$.

Conservation: Consider a node $v \in P$ with e entering v and e' leaving v. There are four cases based on which of e, e' are forward/backward. One case is with e forward and e' backward. Along e, incoming flow to v increases by δ. For e', we subtract δ from $\overline{e'}$, so δ less flow enters v along $\overline{e'}$. So total incoming & outgoing flow stays the same. Likewise for the other three cases.

Also, note that integrality of the weights in our output flow f follows easily from induction. To think about optimality, let's revisit the max-flow / min-cut connection.

Proposition 9.4

For any flow f and s-t cut $(S, \overline{S})$, $\Delta(f) \le c(S, \overline{S})$.

This seems like the kind of thing you don't even need to prove, but we'll prove it later on by showing something even stronger. A consequence is that if we find an f^* and S^* with $\Delta(f^*) = c(S^*, \overline{S^*})$, then this proves both that f^* is a max-flow and $(S^*, \overline{S^*})$ is a min-cut. This is known as an instance of **duality**, which we'll discuss further when we get to integer programming in a few weeks.

Here's another lemma.

Lemma 9.5

For any s-t flow f and any s-t cut $(S, \overline{S})$,

$$\Delta(f) = \sum_{e \text{ out of } S} f_e - \sum_{e \text{ into } S} f_e.$$

The proof is really just 3 minutes of symbol pushing. – David

Proof of Lemma 9.5.

$$\begin{aligned}\Delta(f) &= \sum_{e \text{ out of } s} f_e \\ &= \sum_{v \in S} \left(\sum_{e \text{ out of } v} f_e - \sum_{e \text{ into } v} f_e \right) \\ &= \sum_{e \text{ out of } S} f_e - \sum_{e \text{ into } S} f_e\end{aligned}$$

□

Now it's time to show that our algorithm is optimal. Consider the algorithm at termination,[4] and define S to be the collection of vertices that are connected to v in the residual graph G_f. By our lemma, $\Delta(f) = \sum_{e \text{ out of } S} f_e - \sum_{e \text{ into } S} f_e$. For all e out of S, $f_e = c_e$; otherwise, the other endpoint of e could also be reached in the residual graph. Similarly, for all e into S, $f_e = 0$; otherwise, there would be a backward edge $\overline{e}$ leaving S in G_f. So $\Delta(f) = \sum_{e \text{ out of } S} f_e = c(S, \overline{S})$, meaning f is a max s-t flow and $(S, \overline{S})$ is a min s-t cut.

All that's left are complexity considerations. For now, let's assume the c_e are integral. Then each $\delta \ge 1$ and so each iteration in the `while` loop increases $\Delta(f)$ by at least

[4] We'll see shortly that the algorithm terminates when weights are integral or rational, but not necessarily when weights are irrational. (This begs the question of how the computer is even working with irrationals, but you can invoke an oracle for the sake of just thinking about termination.)

1. Thus the loop terminates. In particular, the number of iterations is bounded by $\sum_{e \text{ out of } s} c_e$. But that's pseudo-polynomial runtime, since the c_e were given to us as numbers (i.e., using $\log(c_e)$ many bits). Furthermore, this pseudo-polynomial behavior can even be materialized with slightly perverse networks. Simply put, the algorithm – as we've specified it – is not efficient.

It can be made polynomial time, however, by using the widest path (largest δ) in each iteration, and strongly polynomial by using shortest path (giving rise to the **Edmonds-Karp** algorithm). If capacities are irrational, then running plain Ford-Fulkerson might not terminate at all. In general, note that the runtime is $\Theta(\widehat{c}(M+N))$, where $\widehat{c}$ is the number of iterations in the while loop. This is a consequence of the fact that each iteration is dominated by a run of BFS or DFS to find a path in the residual graph, which has runtime $\Theta(M+N)$.

§10 Monday, September 26

§10.1 Edmonds-Karp

We'll be analyzing the Edmonds-Karp algorithm today; recall that it's the variant of Ford-Fulkerson which at each iteration selects the shortest path in the residual graph. The goal here is to get from the pseudo-polynomial time of Ford-Fulkerson to true polynomial time. What's the intuition? It's that at each iteration of the algorithm, the shortest paths will never become shorter, and they'll actually become strictly longer "often enough."

More formally:

1. The shortest s-t path never gets shorter.
2. Between any two consecutive saturations of an edge ($f_e = c_e$ or $f_e = 0$), the length of the shortest s-t path strictly increases.

Note that every $2m+1$ consecutive iterations must contain two iterations where the same edge e has $f_e = c_e$ or $f_e = 0$. This is because each iteration saturates at least one edge. Assuming points 1 and 2 above, and noting that the shortest path length can increase at most n times, we have a total of $O(nm)$ iterations and thus a running time of $O(nm^2)$.

Now let's start proving things.

Lemma 10.1

Let P, Q be augmenting paths in iterations $i < j$, with the property that Q pushes flow on some edge e in the opposite direction of P, and for each iteration $i' \in (i, j)$, the flow path $P_{i'}$ does not push flow opposite to either P or Q along any edge. Then $|Q| > |P|$.

Proof. Bit technical. □

Now, assuming the lemma, let's show that the shortest path distance never shrinks. Consider P_r, P_{r+1}. Either P_{r+1} was available in iteration r, in which case it must be no shorter than P_r, or the conditions of Lemma 10.1 are satisfied, and we're done.

Now suppose that e is saturated in iterations $r < r'$. At some point $r'' \in (r, r')$, e must have been used in the opposite direction. We can keep looking at earlier times until the conditions of Lemma 10.1 are met, at which point we're done.

So we're finished with the analysis of Edmonds-Karp!

§10.2 Bipartite matchings II

Recall that we already have an efficient algorithm for finding maximal matchings in bipartite graphs (via Edmonds-Karp on the associated flow problem to a bipartite graph). But there are still lots of natural questions to ask about matchings in bipartite graphs. One is: can we characterize bipartite graphs that have *no* perfect matchings?

Let's say we have $G = (V, E)$ with $V = X \sqcup Y$, $|X| = |Y|$, and $E \subseteq X \times Y$. One obvious obstacle would be to have $S \subseteq X$ with $|S| > |N(S)|$, where $N(S) = \{u \mid \exists v \in S, (u, v) \in E\}$. If such a set S exists, then clearly G can't have a perfect matching.[5]

Is the converse true?

Theorem 10.2 (Hall's theorem)

G has a perfect matching if and only if $|S| \leq |N(S)|$ for all $S \subseteq X$.

Proof. The forward direction is immediate. For the reverse direction, we assume G has no perfect matching and show there is an $S \subseteq X$ with $|N(S)| < |S|$. We'll prove this via max-flow / min-cut. Because G has no perfect matching, the max flow for $\tilde{G}$[6] has value at most $n - 1$. Then there's a min s-t cut $(A, \overline{A})$ of capacity at most $n - 1$.

Now we characterize $c(A, \overline{A})$. Note that there can be no edge from $A \cap X$ to $\overline{A} \cap Y$, because those have capacity ∞. That leaves edges from $Y \cap A$ to t and from s to $X \cap \overline{A}$. There are $|X \cap \overline{A}|$ edges from s to $X \cap \overline{A}$ and $|Y \cap A|$ edges from $Y \cap A$ to t. So $|X \cap \overline{A}| + |Y \cap A| \leq n - 1$.

Now define $S := X \cap A$. Then $|S| = |X \cap A| = n - |X \cap \overline{A}|$. And $N(S) \subseteq Y \cap A$, so $|N(S)| \leq |Y \cap A|$. Then

$$\begin{aligned} n - 1 &\geq |Y \cap A| + |X \cap \overline{A}| \\ n - 1 &\geq |N(S)| + n - |S| \\ |N(S)| &\leq |S| - 1. \end{aligned}$$

So we're done. □

Corollary 10.3

Every d-regular bipartite graph $d \geq 1$ has a perfect matching.

Recall that d-regular bipartite graphs are those in which all nodes have degree exactly d.

Proof of 10.3. We use Hall's theorem. Fix a set $S \subseteq X$. It has $d|S|$ edges sticking out. And at least d unique vertices must be hit by these edges, owing to their degrees being d. □

Furthermore, by repeatedly removing matchings, we can see that d-regular bipartite graphs have at least d disjoint perfect matchings.

Now let's consider the problem of edge-disjoint paths. Given a graph (undirected or directed) with distinguished vertices s, t, the problem is to find a maximum number of s-t paths $P_1, \ldots, P_k$ not sharing any edges. Note that we can reduce to max-flow by giving every edge capacity $c_e = 1$.

[5] A perfect matching would restrict to an injection $S \to N(S)$, but there's a cardinality obstruction to any such injection.

[6] This is the graph associated to G for the flow problem, with edges $s \to X$ and $Y \to t$ of capacity 1, and with edges in E having capacity ∞.

§11 Wednesday, September 28

§11.1 Edge-disjoint paths

Recall the edge-disjoint paths (EDP) problem.

Problem 11.1 (Edge-disjoint paths)**.** Given a graph $G = (V, E)$, undirected or directed, with distinguished vertices $s, t \in V$, find a maximum number of s-t paths not sharing any edges.

We mentioned last time that this reduces to max-flow by giving all the edges in V a capacity of 1 and finding an integral max s-t flow. Then we need to "extract the paths" from this flow. This path extraction step is captured by a path decomposition result.

Lemma 11.2

Given any acyclic (integer) s-t flow f, one can compute – in polynomial time – $k \leq m$ s-t paths $P_1, \ldots, P_k$ with associated (integer) values $a_1, a_2, \ldots, a_k$ such that

$$f_e = \sum_{i \mid e \in P_i} a_i \ \forall e.$$

Proof. We use a greedy algorithm: in each iteration i, find an s-t path p_i where all edges carry flow. Subtract bottleneck amount, call that a_i. Flows stay flows after subtraction, so we can never get stuck early. And $k \leq m$ because each iteration removes an edge from the flow f. □

Lemma 11.3

For any (integer) flow f, there is an acyclic (integer) flow f' with $\Delta(f') = \Delta(f)$ and f' can be obtained efficiently from f.

Proof. We repeatedly remove flow around a cycle. □

So, with these two lemmas, we've made precise our previous step of "extracting paths" in the EDP reduction. To find the paths in the EDP reduction, apply path decomposition lemma to get an integer path decomposition of the integer max-flow f. This gives paths $P_1, \ldots, P_k$ with all $a_i = 1$ (as that's the only integer > 0 and $\leq c_e$). Because $c_e = 1$, the P_i are then disjoint. And k is maximum because otherwise we could get a better flow using $k + 1$ disjoint paths.

Corollary 11.4 (Menger's theorem, edge version)

The maximum number of edge-disjoint s-t paths equals the minimum number of edges whose removal disconnects s and t.

If we replace each instance of "edge" with "vertex" in the previous theorem, we get the vertex version of Menger's theorem.

§11.1.1 Circulations

It turns out that we can naturally generalize the max-flow problem to so-called **circulations**, in which each node v specifies its demand $d(v)$, i.e., how much flow it wants to absorb. (When $d(v) < 0$, then the vertex v is supplying flow.) We must have $\sum d(v) = 0$, otherwise satisfying each v is impossible.

This has an easy reduction to Max-Flow; add a super-source and super-sink, with edges of capacity $-d(v)$, $d(v)$ to/from v.

§11.2 Image / graph segmentation

Problem 11.5. We are a given a graph G with node scores a_v, b_v and edge strengths p_e. Here a_v captures how much v should belong to one side of a partition (foreground, conservative, UCLA, etc.), b_v captures how much v should belong to the other side (background, liberal, USC, etc.), and $p_{(u,v)}$ measures how likely nodes u and v are to agree.

More explicitly, we have the following combined objective function: the score of a partition $(S, \overline{S})$ is

$$Q(S) = \sum_{v \in S} a_v + \sum_{v \notin S} b_v - \sum_{u \in S, v \notin S} p_{u,v}.$$

The goal is to find $(S, \overline{S})$ maximizing $Q(S)$.

We want to find a cut in a graph, so a natural place to start thinking is min-cut. One issue here is that we have positive and negative terms in our objective, so it's not a clear minimization or maximization problem. Fortunately, we can rewrite $Q(S)$ to clean things up a bit:

$$Q(S) = \sum_{v \in V} a_v + \sum_{v \in V} b_v - \Big(\sum_{v \notin S} a_v + \sum_{v \in S} b_v + \sum_{u \in S, v \notin S} p_{u,v} \Big).$$

Now we have $Q(S) = c - R(S)$ for c simply a constant. Then the name of the game is to minimize R. Now, with some thinking, we have a reduction.

We add an s with edges of cost a_v to all $v \in V$. Then we add a t with edges of cost b_v for all v. For any $s-t$ cut, the capacity of the cut is precisely $R(S)$. So the min s-t cut minimizes $R(S)$ and thus maximizes $Q(S)$.

Remark 11.6. This is a pretty general technique whenever you need to find a partition and there are pairwise costs for putting things on opposite sides of the partition. For instance, if two vertices must be on the same side, then you can place an edge between them of infinite capacity.

§12 Monday, October 3

§12.1 NP hardness

We'll begin discussing the notions of computational hardness and impossibility, including NP hardness in particular. Before we get there, though, we need to be more precise about the setup of problems and solutions.

§12.1.1 What is a "problem"?

In the context of impossibility and hardness results, we often study **decision problems**, i.e., those of the form: compute a function $f : \{0,1\}^* \to \{0,1\}$. We often identify the problem with the set $f^{-1}(1) := f^{-1}(\text{"Yes"})$, which is known as a **language**.

Example 12.1

How can we turn max-flow into a decision problem?

Solution. Encode a flow problem in binary, along with a target value F. Have f output 1 if there exists a valid s-t flow f with $\Delta(f) \geq F$, and 0 otherwise. □

Given a solution to 'vanilla' max-flow, we can of course solve this decision version, i.e., by simply computing f^* and comparing $\Delta(f^*)$ with F. Conversely, given a solution to the decision problem we can find $\Delta(f^*)$ with a linear search over $\mathbb{N}$ (or binary search over the capacity of s, if we care about efficiency). The point here is that even these seemingly simple decision problems really do capture the essence and complexity of our original problems. The decision versions are just as hard as the original problems!

Remark 12.2. In general, to convert a maximization problem to a decision problem, add an argument k and ask: Is it possible to get at least k? Likewise for minimization problems (at *most* k?).

§12.1.2 What does it mean to "solve" a problem?

What should it mean for an algorithm to "solve" a problem X, defined by the function f? Well, it should just compute the the function f. (In particular, it will need to halt on every input.)

Definition 12.3 — Let **P** denote the class of all problems X that have a polynomial-time solution. (By polynomial-time we mean there is a polynomial p such that on input x, the algorithm takes time at most $p(|x|)$.)[a]

[a] This is exactly the same as requiring that the algorithm be in $O(p)$ for a polynomial p. Can you see why?

Remark 12.4. What's the model of computation here? We're not going to say much about this, but for now you can assuming it's either the Turing machine or C++. By the extended Church-Turing thesis, it doesn't really matter what we pick.

Definition 12.5 — Let **NP** denote the class of *non-deterministic polynomial time* problems with an efficient certifier. Intuitively, these are problems for which a solution can be *verified* efficiently.

Definition 12.6 — A function $B(x, y)$ is an **efficient certifier** for the language $X \subseteq \{0,1\}^*$ if it has the following properties, for fixed polynomials p, q:

1. If $x \in X$, there exists a y with $|y| \leq q(|x|)$ such that $B(x, y) =$ "Yes".
2. If $x \notin X$, then $B(x, y) =$ "No" for all y.
3. $B(x, y)$ always runs in time $p(|x| + |y|)$.

When $B(x, y) =$ "Yes", we call y a **certificate** for x.

Proposition 12.7
$\mathsf{P} \subseteq \mathsf{NP}$

Proof. Construct the certifier B so that it ignores certificates and solves x outright. □

Definition 12.8 — Let **EXP** denote the class of all problems that have an exponential-time solution, i.e., an algorithm in $O(2^{p(n)})$ for p a polynomial.

Proposition 12.9
$\mathsf{NP} \subseteq \mathsf{EXP}$

Proof. Try all $2^{q(|x|)}$ candidate certificates, and run B on them. This takes time $O\big(2^{q(|x|)} \cdot p(|x| + q(|x|))\big)$. □

P is a pretty organic thing to define, but what about NP? Well, NP consists of all problems whose solutions we can recognize in polynomial time. Then we claim that NP consists of pretty much all problems we could ever hope or want to solve. After all, how could you begin a search for something if you weren't even able recognize what you were looking for?

§12.2 Reductions

So NP contains many problems we'd like to solve. Then we want to know whether doing so efficiently is possible. Is $\mathsf{NP} \subseteq P$? Equivalently, is $\mathsf{P} = \mathsf{NP}$? This has been open for decades – it's the biggest open problem in computer science, and probably in all of math as well. One observation is that if $\mathsf{P} \neq \mathsf{NP}$, then the candidate problems for not being poly-time solvable would be the "hardest" problems in NP. We can use reductions to make this formal.

Definition 12.10 — A **Karp reduction** in polynomial time is a function $f : \{0,1\}^* \to \{0,1\}^*$ such that

1. f runs in polynomial time,[a]
2. if $x \in X$, then $f(x) \in Y$,
3. if $x \notin X$, then $f(x) \notin Y$.

If there is such an f, we write $X \leq_p Y$.

[a] More appropriately, f has a polynomial-time implementation.

Proposition 12.11
If $X \leq_p Y$, then Y is "harder" than X. In particular, if $Y \in \mathsf{P}$ then $X \in \mathsf{P}$, and if $X \notin \mathsf{P}$ then $Y \notin \mathsf{P}$.

Definition 12.12 — A problem X is **NP-hard** if $Y \leq_p X$ for all $Y \in \mathsf{NP}$. A problem X is **NP-complete** if X is NP-hard and $X \in \mathsf{NP}$.

It's pretty easy to see that there are NP-hard problems, but it's not at all obvious that there exist NP-complete problems. Fortunately, they do turn out to exist.

Theorem 12.13 (Cook-Levin)
SAT, and even 3-SAT, are NP-complete.

Lemma 12.14
Reductions are transitive. $\big(X \leq_p Y\big) \wedge \big(Y \leq_p Z\big) \implies X \leq_p Z$.

Proof. Straightforward. □

Corollary 12.15
If X is NP-hard and $X \leq_p Y$, then Y is NP-hard.

§13 Wednesday, October 5

§13.1 Reduction Boogaloo

§13.1.1 Independent Set

The Independent Set problem is that of finding a largest **independent set** in a graph G (i.e., a largest set S of vertices with no edges between any $v, v' \in S$). As usual, we can turn this into a decision problem by asking whether the graph G has an independent set of size at least k.

Theorem 13.1
Independent Set is NP-complete.

Proof. First we show Independent Set is in NP. Easy enough – the certificate is a collection of vertices of size at least k that's independent.

To show that it's NP-hard, we'll need to reduce 3-SAT to it. Namely, we need a function f that maps formulas Φ to instances (G, k) of Independent Set which is efficient and such that $\Phi \in 3\text{–SAT} \iff f(\Phi) \in$ Independent Set. So let's say Φ has n variables $x_1, \ldots, x_n$ and m clauses $C_1, \ldots, C_m$. In 3-SAT, the decision is for each C_j, which literal is designated to satisfy C_j. Of course, we cannot simultaneously pick contradictory literals. In Independent Set, the decision is which nodes to pick; edges prevent us from picking pairs simultaneously.

So we have a reduction: for each clause, generate nodes for its literals. Add edges between literals that are negations of each other, and edges between literals that are in the same clause (so we don't get extra credit for satisfying the same clause "twice"). Finally, set $k = m$, so we indeed satisfy all the clauses.

It's clear that this construction is poly-time. If Φ is satisfiable, pick a true literal from each clause. The corresponding nodes form an independent set of size $k = m$ in G. Conversely, if we grab an independent set in G and set its literals to true, then we satisfy Φ. □

§13.1.2 Vertex Cover

The problem here is that you give me a graph and I try to cover the edges by selecting the minimum number of vertices. (An edge is covered if at least one of its endpoints has been selected.) We again turn this into a decision problem by asking whether this can be done using k or fewer vertices.

Problem 13.2 (Vertex Cover)**.** Given a graph G and integer k, does G contain a **vertex cover** of size at most k?

Theorem 13.3
Vertex Cover is NP-complete.

Proof. It's easy to see that it's in NP– you give me a proposed vertex cover and I check it. That it's NP-hard follows immediately from the following lemma: S is a vertex cover if and only if its complement is an independent set. This is easy enough to prove once you know to think about it. So the reduction is just $(G, k) \mapsto (G, n - k)$. □

§13.1.3 Set Cover

Problem 13.4 (Set Cover)**.** Given a universe U of n elements and sets $S_1, \ldots, S_m \subseteq U$, determine whether there is a **set cover** of size k, i.e., k subsets $S_{j_1}, \ldots, S_{j_k}$ such that $\bigcup_i S_{j_i} = U$.

> *This problem pops up everywhere. And like any problem that pops up everywhere, it's NP-hard.* – David.

Theorem 13.5
Set Cover is NP-complete.

Proof. This is in NP; you give me the collection of subsets and I check that they cover. Also Vertex Cover reduces to it pretty easily; you have one set for each vertex, consisting of the edges that it touches. □

§13.1.4 3-Dimensional Matching

Problem 13.6 (3-Dimensional Matching)**.** Given sets A, B, C of size n, along with a collection $X \subseteq A \times B \times C$ of triples, is there a subset $T \subseteq X$ that forms a perfect **3-dimensional matching**?[7]

This simultaneously has the flavor of a packing and a covering problem:

1. Packing: Can we find n disjoint triples?
2. Covering: Can we cover all elements with n triples?

[7] i.e., each $e \in A \cup B \cup C$ is in exactly one triple in T.

Theorem 13.7
3-Dimensional Matching is NP-complete.

Proof start. It's easy to see that it's in NP. Now we reduce from 3–SAT. We'll encode truth values of variables in choices of triples. Because variables can occur in multiple clauses, we need "consistency gadgets" to ensure that they always take the same value. We'll pick up here next time. □

§14 Monday, October 10

§14.1 Reduction Boogaloo II

We started off by completing the proof of NP completeness for 3-Dimensional Matching. It was fairly involved, and I didn't write it down.

§14.1.1 Subset-Sum

Recall the Subset-Sum problem: given $a_1, \ldots, a_n \in \mathbb{N}$ and $b \in \mathbb{N}$, is there an $S \subseteq \{1, \ldots, n\}$ with $\sum_{i \in S} a_i = b$? We've already shown that it reduces to Knapsack, though that doesn't buy us much right now.

Theorem 14.1
Subset-Sum is NP-complete.

Proof. This is in NP – you give me S and I check that it works. Now we reduce from 3DM, or rather X3C (i.e., given a universe U of size $3n$ and m triples in U, is there an exact covering of U?). To show X3C $\leq_p$ Subset-Sum, fix a universe U with triples $T_1, \ldots, T_m$. For each triple T_j, generate an integer a_j that has 1 in exactly the digits i such that $x_i \in T_j$. (So each a_j has exactly 3 non-zero digits.) And the target number b consists of $3n$ ones. By interpreting numbers in base $m+1$, we can ensure no carries and thus that this works in both directions. □

§14.2 Linear programming

Definition 14.2 (Linear programming) — Given a matrix $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^n$, find a solution to $Ax \leq b$, entry-wise. That is, find $x_1, \ldots, x_n$ so that

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n &\leq b_1 \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n &\leq b_2 \\ &\vdots \\ a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n &\leq b_m \end{aligned}$$

Of course, this looks very similar to actually *solving* systems of linear equations. In fact, it subsumes solving systems of linear equations, by writing $Ax \leq b$ and $(-A)x \leq -b$ in the same matrix. More generally, by multiplying any inequality with -1, we can mix $\geq$ and $\leq$ in Definition 14.2. We can also generalize even further to maximize/minimize $c \cdot x$

subject to $Ax \le b$. This is done by performing binary search for optimal values of $c \cdot x$, with the additional constraint that $c \cdot x \le \gamma$ on the decision LP.[8]

Note that the decision version of LP (given A, b is there an x with $Ax \le b$?) is in NP. Our certificate is x; this actually makes use of the non-trivial fact that polynomially many digits in x will be enough, so a certificate has polynomial length.

Next time, we'll use duality to show that LP is in co-NP.

§15 Wednesday, October 12

§15.1 Linear programming II

Last time we learned about linear programming – now that we have a new problem, the most natural question to ask is whether it can be efficiently solved (i.e., in polynomial time). This was an open question for more than a decade, but it was eventually resolved in the positive with the *ellipsoid method*. Another method, which is very frequently used, is the *simplex method*. It turns out to have exponential time in the worst case, but it's quite fast in practice.

We won't delve into the details of any of these methods right now – see CS 675 for more – but it'll be useful at least to know that there's a black-box method for efficiently solving linear programs.

Example 15.1 (Diet problem)

Let's say there are 3 nutrition groups: sugar, vitamins, and protein. There are also 3 foods: ramen, pizza, and chicken nuggets. We have a matrix A recording the nutrition information of each food: a_{ij} units of nutrition i per unit of food j. So $a_{p,r}$ is the amount of protein in ramen, $p_{s,p}$ is the amount of sugar in pizza, and so on.

Let's also say we have costs per unit for each of the foods. Now minimizing our grocery bill subject to some nutrition requirements on total sugar, vitamins, and protein is a linear program. So we can solve it with an LP solver.

Example 15.2 (Maximum s-t flow)

We have variables f_e for each edge. Then the axioms for non-negativity, capacity, and conservation are all linear. Our objective is also linear, so this is a linear program!

Remark 15.3. We don't have the integrality guarantee of Ford-Fulkerson with this solution to max-flow! Can we somehow add a constraint to our LP solver that requires that the solution be integral? Probably not. Linear programming with the requirement that solutions be integral is NP-complete. After all, you can pretty easily reduce things like vertex cover to integral LP (by imposing the requirement that your variables are in $\{0, 1\}$).

Sometimes – perhaps even often – when solving combinatorial problems, it is convenient or necessary to have many constraints (i.e., exponentially many, even an infinite number). Clearly this is problematic. However, such LPs can sometimes be solved in polynomial time; in particular, when you have the following two:

1. A poly-time algorithm for deciding whether a proposed solution x satisfies all constraints (i.e., a "membership oracle").

[8] This feels a bit bogus to me.

2. A poly-time algorithm for finding a violated constraint if one exists (i.e., a "separation oracle").

This isn't just a nice little aside: it's actually a really important theorem.

§15.2 LP duality

Let's motivate and arrive at LP duality using an example. Say we have an LP problem of the form:

$$\begin{aligned} \text{minimize } & 3x_1 + 4x_2 + 2x_3 \\ \text{such that } & x_1 + 2x_2 + x_3 \geq 3 \\ & 0x_1 + x_2 + 2x_3 \geq 4 \\ & 3x_1 + x_2 + x_3 \geq 5 \end{aligned}$$

To get a lower bound on the optimum, we can multiply rows with $\alpha_1, \alpha_2, \alpha_3$ so long as the sum of scaled coefficients is at most the coefficient in the objective. To get the best lower bound, we want to maximize the sum of scaled right-hand sides.

So our new problem – to get the best possible lower bound – is

$$\begin{aligned} \text{maximize } & 3\alpha_1 + 4\alpha_2 + 5\alpha_3 \\ \text{such that } & 1\alpha_1 + 0\alpha_2 + 3\alpha_3 \leq 3 \\ & 2\alpha_1 + 1\alpha_2 + 1\alpha_3 \leq 4 \\ & 1\alpha_1 + 2\alpha_2 + 1\alpha_3 \leq 2 \end{aligned}$$

So this process gives us a new linear program, called the **dual linear program** of the original one. Notably, minimization has turned into maximization, our objective and our constraints switched, and our matrix of linear coefficients became its transpose. Then taking duals twice recovers our original **primal linear program**.

Theorem 15.4 (Weak duality)

For all x feasible for the primal LP and all y feasible for the dual LP, $c \cdot x \geq b \cdot y$.

Proof. Let's use $\cdot$ for inner products and $\times$ for matrix multiplication. We have:

$$\begin{aligned} c \cdot x &= c^T \times x \\ &\geq (A^T \times y) \cdot x \\ &= (y^T \times A) \times x \\ &\geq y^T \times b \\ &= b \cdot y. \end{aligned}$$

Notably, we made use of nonnegativity of x and y in each inequality. □

Theorem 15.5 (Strong duality)

$$\min_{\substack{Ax \geq b \\ x \geq 0}} c \cdot x = \max_{\substack{A^T y \leq c \\ y \geq 0}} b \cdot y.$$

Proof. Take CS 675 :) □

§16 Monday, October 17

Last time we talked about linear programming (LP) and about duals. Informally, duals are like proof that you can't do better in your original, or *primal*, LP. We also mentioned that under a unimodularity condition on matrices, LPs are guaranteed to have some integral solutions (though an LP solver may not be guaranteed to return an integral solution).

§16.1 Approximation algorithms

Let's begin discussing **approximation algorithms**: the motivation here is always that we don't have the resources to solve the problem outright (i.e., optimally). As we've mentioned, NP-hard problems show up all over the place, giving us good reason to search for efficient approximations. (Or you might even have a cubic-time solution and be willing to approximate in order to get something like loglinear time.)

Here's a subtle point: in order to even speak about approximating a problem like 3-SAT, we would need to cast it from a decision problem back to an optimization problem (i.e., where the goal is to satisfy as many clauses as possible).

> *Approximation is really only meaningful if you're trying to maximize or minimize a quantity. If it's just a yes/no question, approximation doesn't mean anything.* – David.

Now let's formalize approximation.

Definition 16.1 — For a minimization problem with objective c, an algorithm A is an **α-approximation** ($\alpha \geq 1$) if for all inputs I,

$$c(A(I)) \leq \alpha \cdot c(\mathsf{OPT}(I)).$$

Notably, we can allow α to be a function of $|I|$, like $\alpha = \log(|I|)$.

Implicitly, we'll be interested in polynomial-time approximation algorithms. In principle, you could be excited about a singly exponential approximation to problem that's 2-EXP-hard, but we won't really be discussing cases like this.

Remark 16.2. A difficulty in analyzing approximation algorithms is that characterizing α requires knowing something about OPT. But if you really knew a lot about OPT, you could probably just solve the problem to begin with. So often times you need two steps here: getting a good bound on OPT, and then showing that your algorithm isn't too much worse than that.

Usually, algorithm design actually starts from a bound on OPT and then derives an algorithm from that bound. Coming up with a good bound on OPT is often half the battle.

Remark 16.3. Why did we take $\alpha \cdot c(\mathsf{OPT}(I))$, rather than $\alpha + c(\mathsf{OPT}(I))$? One reason is that we want our guarantees to be independent of the *units* that we use to measure our objective. Furthermore, you might have a problem in which this additive α cannot be bounded at all (i.e., does not exist), even though you have a very natural multiplicative α of 2. There are some relatively rare exceptions, though, in which you want an additive factor, rather than (or perhaps along with) a multiplicative one.

Example 16.4 (Vertex Cover)
Given an undirected graph G, the objective is to cover all edges using as few vertices as possible. Find an approximation algorithm.

Proof. As discussed, we begin with a lower bound on the optimal solution. Here's a lemma: if M is a matching in G (i.e., disjoint collection of edges), and S is a vertex cover, then $|M| \le |S|$. Proof: to be a vertex cover, S must contain at least one endpoint for each $e \in M$. And no vertex is an endpoint of two edges in M, as M is a matching.

So we have our lower bound. And this gives rise to an approximation algorithm: find a maximal matching M, and for each edge $e \in M$, select both endpoints. Call these endpoints S. This is a vertex cover: by maximality of M, each edge must have one of its endpoints in S, otherwise we could add it to M. So then we have

$$|S| = 2|M| \le 2 \cdot \mathsf{OPT}.$$

Now how do we get a maximal matching in G? Just be greedy! Maximality is a local condition, so we don't need to be that clever: pick disjoint edges until you can't anymore. □

Remark 16.5. Some context here: no algorithm with better than a 2-approximation is known, and beating a 7/6-approximation is NP-hard.

§16.1.1 (Metric) Traveling Salesman

Problem 16.6. Given an $n \times n$ distance matrix $d_{i,j}$ inducing a metric on $\{1, \dots, n\}$, find a tour — starting at some $v \in \{1, \dots, n\}$, visiting all points and returning to v — of minimum total length.

Proposition 16.7
TSP is NP-complete.

Proof. This is in NP – you give me the tour and I check that its total distance is less than the target value k.[9] To see that it's NP-hard, you can reduce from Hamiltonian cycles, which we haven't discussed. □

Remark 16.8. Euclidean TSP is the setting in which points are explicitly given as d-dimensional (say, integer) coordinates, whose distances are the usual Euclidean distances. Even this is NP-hard, but it's not actually known to be in NP! The issue here has to do with irrational distances, and the number of digits you need to compute to compare a sum of irrationals with an integer. So this is a bit annoying, but it's worth mentioning as a case in which something is "obviously" in NP but turns out to perhaps not be!

So, in the search of an approximation algorithm for TSP, let's again think of a lower bound for the optimum. Some obvious candidates are "longest edge in G" or "sum of the cheapest $(n-1)$ edges in G" or "radius of G", but those are too local. You can maliciously create examples where these give terrible lower bounds for the optimum by

[9] Since we're considering NP-completeness, we need to move back to the decision version of TSP!

adding local features to your graph that makes this bound very low and making the rest of the graph gnarly.

A more global lower bound would be to look at an MST!

Lemma 16.9
$c(MST) \leq c(\mathsf{OPT})$.

Proof. OPT itself is a spanning collection of edges in G. □

Algorithm 16.10 — Compute an MST for G. To get a tour, perform DFS on the MST and use the triangle inequality to shortcut out any repeated nodes.

Analysis. DFS visits each edge twice, and shortcutting only reduces the cost, due to the triangle inequality. So the cost of our solution is $\leq 2 \cdot c(MST) \leq 2 \cdot c(\mathsf{OPT})$, and it's a 2-approximation. □

Proof. We can do better than a 2-approximation. The Cristofides algorithm gets a 1.5 approximation, and more recently something like a 1.49999999 approximation was discovered. Furthermore, there is a constant $c > 1$ such that metric TSP cannot be approximated within c unless P = NP. □

§16.1.2 (Metric) Steiner Tree

Problem 16.11. We are given n points with metric distances. k of them, forming a set T, are *terminals*. The remaining $n - k$, forming X, are *Steiner nodes*. Now build a tree spanning all of T – and optionally some/all of X – of minimum total cost.

An obvious algorithm is to do an MST on T. This turns out to give a 2-approximation.

Proof. Take OPT, do DFS (which at most doubles the cost), then shortcut repeated nodes. Now, all nodes – and thus all Steiner nodes – have degree 2, and so the Steiner nodes can be removed without increasing cost, owing to the triangle inequality. So this gives us a spanning tree of T with cost $\leq 2 \cdot \mathsf{OPT}$. So the MST is a 2-approximation. □

§17 Wednesday, October 19

§17.1 Approximation algorithms II

§17.1.1 Set Cover and Maximum Coverage

We'll be discussing Set Cover and Maximum Coverage today. Each involves a universe U of elements and collection of sets $S_1, \ldots, S_m \subseteq U$. For Set Cover, the sets have associated costs c_j, and the goal is to cover all of U at minimum total cost. In Maximum Coverage, the elements of U themselves have values v_i. The goal is to select k sets that maximize the sum of all selected values in U (without double counting).

Let's look at Set Cover first; we'll actually begin by specifying the (greedy) approximation algorithm, and analyze it afterwards.

Algorithm 17.1 — Start with $R = U$, no sets selected. While $R \neq \varnothing$, select S_j

minimizing $\frac{c_j}{|R \cap S_j|}$ and set $R = R \setminus S_j$.

How can we show that this greedy algorithm doesn't pay very much? If you simply consider our solution compared to the optimal solution, the task of comparing the two can seem pretty daunting. Let's try to break this up a bit. One observation/intuition is that if our algorithm pays a lot for some set, that set must contain expensive-to-cover elements, so OPT should have to pay a lot to cover these elements as well.

In particular, we will assign a price p_e to each element $e \in U$, then show that the cost of *any* set cover can be lower-bounded in terms of the p_e.

Analysis of Algorithm 17.1. Say that in iteration k we pick the set S_k. Then we define, for all $e \in S_k \cap R$, $p_e = \frac{c(S_k)}{|R \cap S_k|}$. Then $\sum_{j \in J} c_j = \sum_{e \in U} p_e$. Now we make use of a lemma.

Lemma 17.2

For any set S_j — in particular, even those not in our chosen collection — we have that

$$c_j \cdot H(|S_j|) \geq \sum_{e \in S_j} p_e,$$

where $H(k) = \sum_{i=1}^{k} \frac{1}{i}$.

Proof. Let $S_j = \{1, \ldots, k\}$, such that our greedy algorithm covers elements in the order $1, 2, \ldots, k$ (some at the same time). Now consider an arbitrary element i. At the point when it is first covered by i, we know that elements $\{i+1, \ldots, k\}$ haven't been covered. Now let's say our greedy algorithm picked the set S_ℓ at this point, which minimized $\frac{c_\ell}{|R \cap S_\ell|}$. One option for S_ℓ was S_j itself, which would have given a cost of at most $\frac{c_j}{k-i+1}$. So

$$p_i = \frac{c_\ell}{|R \cap S_\ell|} \leq \frac{c_j}{|k - i + 1|}.$$

And thus,

$$\sum_{i \in S_j} p_i \leq c_j \sum_{i=1}^{k} \frac{1}{k - i + 1} = c_j \cdot H(k).$$

□

Now let $\mathcal{C}^*$ be an optimal solution. Using the previous lemma, we have:

$$\begin{aligned}
\text{cost}(\mathcal{C}^*) &= \sum_{j \in \mathcal{C}^*} c_j \\
&\geq \sum_{j \in \mathcal{C}^*} \frac{1}{H(|S_j|)} \sum_{e \in S_j} p_e \\
&\geq \min_{j \in \mathcal{C}^*} \frac{1}{H(|S_j|)} \cdot \sum_{j \in \mathcal{C}^*} \sum_{e \in S_j} p_e \\
&\geq \min_{j \in \mathcal{C}^*} \frac{1}{H(|S_j|)} \sum_{e \in U} p_e.
\end{aligned}$$

So our algorithm is a $\max_{j \in \mathcal{C}^*} H(|S_j|)$-approximation. In particular, that means it's an $H(|U|) \approx \ln(|U|)$-approximation. □

Remark 17.3. Unless P=NP, you cannot get a $(1-\epsilon)\ln(n)$ approximation in polynomial time, for any $\epsilon > 0$.

Now to Maximum Coverage – it's straightforward to design a similar greedy algorithm.

Algorithm 17.4 — For k iterations, pick the S_j giving the largest added value (i.e., value over elements that have not already been selected). That is,

- $T_0 = \varnothing, R_0 = U$
- in each iteration $t = 1, \ldots, k$:
  - let $j(t)$ be the index of the set maximizing $\sum_{i \in S_j \cap R_{t-1}} v_i$
  - set $T_t = T_{t-1} \cup \{j(t)\}$ and $R_t = R_{t-1} \setminus S_{j(t)}$

The intuition behind our analysis will be as follows: in each iteration in which the value of T_t is far from the value of OPT, the algorithm makes "good progress."

Analysis of Algorithm 17.4. Let OPT be the total value attained by the optimal solution and V_t be the total value in our greedy algorithm's chosen sets after t iterations. Then

$$V_1 = V_1 - V_0 \geq \frac{1}{k}\mathsf{OPT},$$

since one of the sets in OPT must contribute at least a $\frac{1}{k}$ fraction of the value. More generally, in iteration t: there exists a $j \in \mathcal{C}^*$ such that the value of $T_t \cup \{j\}$ is higher than that of T_t by at least $\frac{\mathsf{OPT}-V_t}{k}$. (In particular, by adding all of $\mathcal{C}^*$ to T_t, we would get least OPT value.)

Thus at least one set in $\mathcal{C}^*$, and thus in the world, adds a value of $\frac{\mathsf{OPT}-V_t}{k}$. Since our algorithm is greedy, we have

$$V_{t+1} - V_t \geq \frac{\mathsf{OPT} - V_t}{k}.$$

Rewriting gives us $V_{t+1} \geq (1-\frac{1}{k}) \cdot V_t + \frac{1}{k} \cdot \mathsf{OPT}$. Recalling $V_0 = 0$, we can prove by induction that $V_t \geq (1 - (1-\frac{1}{k})^t) \cdot \mathsf{OPT}$. So this is better than a $(1-\frac{1}{e})$-approximation, since $(1-\frac{1}{k})^k \to \frac{1}{e}$ from below. □

In fact, our work here can be generalized to a wider result.

Theorem 17.5 (Nemhauser-Wolsey-Fischer)
Let $F : 2^U \to \mathbb{R}$ be a function such that

- $f(S) \geq 0 \ \forall S$ (non-negative),
- $f(S) \geq f(T)$ for $S \supseteq T$ (monotonicity),
- $f(S \cup \{x\}) - f(S) \geq f(T \cup \{x\}) - f(T)$ whenever $S \subseteq T$ (**submodularity**).

Then the greedy algorithm for maximizing $f(S)$ subject to $|S| \leq k$ is a $(1-\frac{1}{e})$-approximation.

Proof. The proof carries over almost identically. The only thing we need to show is that if f is submodular, then for all $T \subsetneq T'$, there is a $j \in T' \setminus T$ with

$$f(T \cup \{j\}) - f(T) \geq \frac{1}{|T' \setminus T|}\big(f(T') - f(T)\big).$$

This only takes a couple minutes to show, but we're out of time for today, so we'll pick up here next time. □

§18 Monday, October 24

§18.1 Approximation algorithms III

Here's the key result for Nemhauser/Wolsey/Fischer that we mentioned last time. It's a very popular tool.

Lemma 18.1

Let $f : 2^U \to \mathbb{R}$ be monotone submodular and $S \subsetneq T$. Then there exists an $e \in T \setminus S$ with

$$f(S \cup \{e\}) - f(S) \geq \frac{1}{|T \setminus S|}\big(f(T) - f(S)\big).$$

Proof. Straightforward contradiction. Or, more directly, let $T \setminus S = \{e_1, \ldots, e_k\}$ in some order, and let $S_i = S \cup \{e_1, \ldots, e_i\}$. Then

$$\begin{aligned} f(T) - f(S) &= \sum_{i=1}^{k} f(S_i) - f(S_{i-1}) \\ &\leq \sum_{i=1}^{k} f(S \cup \{e_i\}) - f(S). \end{aligned}$$

So at least one term must be $\geq \frac{1}{k}\big(f(T) - f(S)\big)$. □

Here are some interesting related directions to think about:

1. What if we cannot evaluate f precisely, but only approximate it to precision ϵ?
   - Here, and often in other settings, the loss in approximation can be bounded in terms of ϵ. This kind of imprecision can be much worse if it occurs in your *constraint* rather than your objective. E.g., if you overestimate the cost of a set and so you don't buy it, though it would've been really good.
2. We get a $(1 - \frac{1}{e})$-approximation for submodular maximization for the k-uniform matroid (via our greedy algorithm). The same algorithm — which we would then call Kruskal — gives an optimal solution for *any* matroid when f is linear.
3. How about monotone submodular functions on *arbitrary* matroids?
   - By an older theorem of Nemhauser, greedy gets a 1/2-approximation on arbitrary matroids for monotone submodular maximization.
   - By a theorem of Vondrak, "continuous greedy" (which we won't define here) is a $(1 - \frac{1}{e})$-approximation for maximizing a monotone submodular function subject to *any* matroid constraint.[10]

[10] This is a truly beautiful result about which you can learn more in CS 675.

§18.1.1 Max-cut

Problem 18.2. Given an undirected graph $G = (V, E)$, partition $V = S \cup \overline{S}$ to maximize the number of cut edges.

There are many possible algorithms here. It turns that a silly one that works is to flip a coin for each vertex $v \in V$ and place it in S if it lands heads. So you don't even look at your edge set! Right now we'll analyze a **local search** algorithm.

Algorithm 18.3 (Local search) — Start with an arbitrary $S \subseteq V$. While moving a vertex from S to $\overline{S}$ or vice versa improves the solution, perform such a move.

This algorithm has at most $|E|$ iterations, since the number of cut edges goes up by at least one in each iteration.

Remark 18.4. What if each edge e has weight w_e and the goal is to maximize the total weight cut? We now get a pseudo-polynomial guarantee on runtime, and this is pseudo-polynomial even if we choose the biggest improvement each time.

To get our approximation guarantee, note that at termination, for each vertex v, at least as many edges incident to v cross the cut as are inside S or $\overline{S}$. So at least half its incident edges cross the cut. And thus at least half of all edges cross the cut. So we have a 2-approximation.

This analysis might not be tight, though. One key observation is that our upper bound on OPT is tight, since there are indeed some graphs where you can cut all edges. And our lower bound is also tight, since there are graphs — like K_n — where you can cut at most $\frac{1}{2} + o(1)$ many edges. So we'll never characterize *any* algorithm as being better than a $1/2$-approximation unless we get finer with our analysis. And in fact there indeed exists an ≈ 0.878-approximation using rounding on a semi-definite program.

§18.1.2 Load balancing

Problem 18.5. We are given n jobs with running times t_i and access to m machines. Divide the jobs over the machines to minimize their *makespan*, i.e., the latest finish time of any job. Stated differently, partition $\{1, \dots, n\}$ into $S_1, \dots, S_m$ to minimize $\max_j \sum_{i \in S_j} t_i$.

We've already seen that this is NP-hard even when $m = 2$! So, as expected, we'll need an approximation algorithm.

Algorithm 18.6 — Go through the jobs in an arbitrary order. In iteration i, add job i to the set S_j minimizing $\sum_{i \in S_j} t_i$.

Let's analyze this. Clearly $\mathsf{OPT} \geq \frac{1}{m} \sum_i t_i$. This isn't enough, though – it's extremely loose when we have one job and m machines. So we'll also use that $\mathsf{OPT} \geq \max_i t_i$. Now let S_1 be the set of largest sum in our greedy algorithm. Before the last job i was added to S_1, it was the lowest loaded. So,

$$\sum_{i' \in S_1 \setminus \{i\}} t_{i'} \leq \sum_{i' \in S_j} t_{i'}, \qquad \forall j \neq 1.$$

Then $\sum_{i' \in S_1 \setminus \{i\}} t_{i'} \leq \frac{1}{m} \sum_{i'} t_{i'} \leq \mathsf{OPT}$. And the last item we add to S_1 is at most OPT, so we have a 2-approximation.

This analysis is tight: our algorithm can actually be off by a factor of $2 - o(1)$ from OPT. An improvement to this algorithm is to process jobs from largest to smallest (rather than in an arbitrary order) which gives rise to a 3/2 optimization. We need better bounds on OPT to prove this, though — otherwise $m + 1$ jobs with $t_i = 1\ \forall i$ gives a problem for the analysis.

Remark 18.7. As we've mentioned repeatedly, good bounds on OPT are key to both designing and analyzing approximation algorithms!

§19 Wednesday, October 26

§19.1 Approximation algorithms IV

Today we'll be talking about LP relaxations of integer programs. The general idea of LP-based approximation algorithms is as follows:

1. Write the optimization problem as an integer LP.
2. Remove the integrality constraint.
3. Solve the resulting LP.
4. Do some clever post-processing of the solution to get an integer solution.

The last step is known as the **rounding** of the LP, and it's usually where the heart and creativity of the algorithm really lie. Of course, even though we're calling it rounding, we'll rarely simply be rounding each entry to its nearest integer – we'll often have to be more clever than that. The analysis of such an approximation algorithm usually takes the following form for minimization problems.

1. Observe that $\mathsf{OPT}_{LP} \le \mathsf{OPT}_{IP}$.
2. Prove that the cost of our rounded solution is $\le \alpha \cdot \mathsf{OPT}_{LP}$.
3. Conclude that we have an α-approximation!

§19.1.1 Weighted Vertex Cover

The setup is that of Vertex Cover, but vertices have costs c_v and we want a vertex cover of minimum total cost. So let's cast this as an LP. Our variables are x_v, which is 1 if v is selected and 0 otherwise. Our goal is to minimize $\sum_v c_v x_v$ subject to $x_u + x_v \ge 1$ for all edges $e = (u, v)$. In the true integer program (IP), we'd demand that $x_v \in \{0, 1\}$, but we'll remove it for the sake of our LP relaxation.

With the integrality constraint removed, we can efficiently solve the fractional LP and get a solution. Obviously, the natural intuition here is that larger values in our LP solution should probably be cast to 1 and smaller values are more likely to be cast as 0. So the most obvious algorithm is to simply round each x_v to the nearest integer.

Algorithm 19.1 — Solve our LP relaxation of the integer program for Weighted Vertex Cover, and round each variable x_v to 1 if $x_v \ge 1/2$ and 0 otherwise.

Analysis. We should first show that this outputs a vertex cover. Note that our LP has the constraint that $x_u + x_v \ge 1$ and that $x_v \ge 0$. So for each edge, at least one vertex will

be rounded to 1. Furthermore, the cost of our solution is at most twice that of the LP optimum, since we buy at most twice as much of each vertex as the LP optimum does. So we have a 2-approximation. □

How could we prove that there's no better way of rounding this particular LP? We could come with a particular instance in which the solution to our LP is at least twice as good as the solution to the ILP. This is known as measuring the *integrality gap* of our LP with respect to the ILP.

Definition 19.2 — The **integrality gap** of an LP relaxation of an ILP is the worst-case ratio of $\frac{\mathsf{OPT}_{ILP}}{\mathsf{OPT}_{LP}}$. If the analysis of an LP rounding algorithm only compares the algorithm's output to OPT_{LP} — which is common — then it cannot achieve an approximation guarantee better than the integrality gap.

Proposition 19.3

Our previous LP has an integrality gap ≥ 2, so we can't round it better without a more sophisticated analysis.

Proof. Look at K_n, the complete graph on n vertices, and give all vertices cost 1. Then the best integral solution has cost $n-1$. In particular, any choices of $n-1$ vertices works, and if you miss out on 2 vertices, then they have an edge between them that you miss. But you can solve this in the fractional LP by buying $1/2$ units of each vertex, which has total cost $n/2$. So the integrality gap is at least

$$\sup_{n \in \mathbb{N}} \frac{n-1}{n/2} = 2.$$

□

§19.2 Load balancing II

Let's revisit load balancing under the additional restriction that certain jobs can only be completed by certain machines. So we have a bipartite graph specifying which machines can solve each job.

Let's cast this as an ILP. We'll have variables x_{ij} which are 1 if job i goes on machine j and 0 otherwise. Now our constraints are that $\sum_j x_{ij} = 1 \ \forall i$, $x_{ij} \in \{0,1\}$, $x_{ij} \geq 0$. How do we write our objective? There's a wrinkle here, which is that maximums aren't linear, but there's a way out. We add a new variable L to reflect the maximum load and add constraints on it of the form $L \geq \sum_i x_{ij} \cdot t_i \ \forall j$. Now our objective is to minimize L.

Remark 19.4. This is a fairly common trick for incorporating maximums and minimums in linear programs. This works well when you want to maximize a minimum, or vice versa, but not so much if you're maxing a max or minimizing a min.

There's an issue with our LP relaxation, though. It has a huge integrality gap of m, the number of machines, because one job could be split fractionally over m machines. So let's add the constraint to our fractional LP that $L \geq t_i \ \forall i$. This constraint is redundant for the ILP, but it restricts the LP solution in a really useful way. This is a common technique, known as **strengthening** an LP.

Now let's get to the rounding part. Our fractional solution might have many positive x_{ij} values. To clean it up, we can remove cycles. In particular, consider the bipartite

graph of jobs and machines with an edge (i, j) when $x_{ij} > 0$. While this graph contains a cycle, we can pick any such cycle and divide edges by alternating into sets S^-, S^+. Set $\epsilon = \min_{e \in S^-} x_e$. For all $e \in S^-$, we decrease x_e by ϵ and for all $e \in S^+$ we increase x_e by ϵ.

As we started with a cycle, this keeps all loads unchanged – so the solution doesn't get worse – and now at least one additional x_{ij} becomes 0. By repeating, we obtain a solution in which there are no cycles. We now have that the graph is a forest.

As a second step, for each tree in the resulting forest, we process as follows:

1. Pick an arbitrary node, and root the tree there.
2. For each job that is a leaf, assign it to its unique parent.
3. For each job that is an internal node, assign it to one of its children.

Now consider the load on a given machine j. It's equal to the load from the children of j that are leaves and possibly the load of j's parent. This first sum is at most L, since the edge weights from j to its children i must be 1. In particular, those jobs i are not split up among any machines, as they're leaves. And the load from the parent of j adds at most a cost of L to this load. So our algorithm is a 2-approximation.

Now that we've found an approximation algorithm using LP relaxation and rounding, it actually turns out that we don't need to solve the LP explicitly! We can instead binary search for the correct L value. For each guess, we build a flow network with edges (s, i) of capacity t_i, edges (i, j) of capacity ∞, and (j, t) of capacity L. Then we see if we can get a flow of $\sum t_i$ in this network.

§20 Monday, October 31

§20.1 Online algorithms

We'll be spending the next couple lectures on **online algorithms**, which are those that receive input in a sequence of installments rather than all at once. Furthermore, after the arrival of each chunk of data, decisions must be made irrevocably on the fly, without knowledge of future input parts.

Example 20.1

You want to rent out your machine to maximize profit, but you don't know future job offers, and you must reject or accept the current ones in real time.

Some applications include traffic routing, emergency response, network design (you'd like to keep your nodes connected, but don't know where future nodes will arise), and data structures (as you don't know which items will be accessed in the future).

An immediate observation is that any online learning algorithm will in general make suboptimal decisions compared to having hindsight. This is just the nature of the game. As with approximation algorithms, we'd still like to quantify the quality of our imperfect solution. This is where the **competitive ratio** comes in: it is the ratio of how much worse our algorithm does compared to the optimum solution in hindsight.

Definition 20.2 — An algorithm A is called *c-competitive* if there exists an α such that for all inputs I,

$$\text{cost}(A(I)) \leq c \cdot \text{cost}(\mathsf{OPT}(I)) + \alpha.$$

And A is *strictly c-competitive* if this holds with $\alpha = 0$.

Usually α should be independent of the number of time steps involved in I, but may be allowed to vary with respect to the problem's underlying size (e.g., number of machines to rent, number of emergency robots to send out, etc.). More explicitly, problems often have 2 natural parameters, as in # robots and # requests. Usually one stays bounded while the other grows. Constants α, c should only depend on the bounded parameter if at all.

Definition 20.3 — The **competitive ratio** of A is the smallest c such that A is c-competitive.

§20.1.1 Ski rental

Here's the problem: each day, you decide whether to rent or buy a pair of skis. Renting costs R and buying costs B. After each day of skiing, you find out if you quit permanently or go at least one more time.

Offline, this is trivial. Online, one observation is that there's really only one kind of strategy: rent for d many days and then buy. And as soon as we buy the skis, we know that our adversary will have us quite skiing. So we'll end up skiing for $n := d + 1$ many days.

Furthermore, the optimum cost for skiing over n days is $\min(B, n \cdot R)$. And our algorithm's cost is $B + (n-1) \cdot R$. So the competitive ratio here is

$$\frac{B + (n-1)R}{\min(B, nR)}.$$

We're restricting focus to $n = d + 1$, as this is the worst case for our algorithm. Now we want to find n minimizing this. We have

$$\begin{aligned} R &= \max\left(\frac{B + (n-1)R}{B}, \frac{B + (n-1)R}{nR}\right) \\ &= 1 + \max\left((n-1)\frac{R}{B}, \frac{B-R}{nR}\right). \end{aligned}$$

This is minimized when the terms in the max are equal, as one term increases with n and the other decreases with n. And this occurs when $n = \frac{B}{R}$, so we have our strategy. Rent until you would have paid the buying price with the next rental, then buy. And this gives us a competitive ratio of

$$\frac{Rn + (n-1)R}{Rn} = 2 - \frac{1}{n} = 2 - \frac{R}{B}.$$

Here, we've assumed an adversary who can observe all of the algorithm's choices. Traditionally, we assume an adversary who designs an input ahead of time. The additional assumption we're making here is not too strong for deterministic algorithms, but it would be for randomized algorithms.

In fact, if your algorithm gets to flip random coins, it becomes important to distinguish between adversaries that know about your coin flips ahead of time and adversaries that don't. With randomization — and adversaries that don't know about coin flips ahead of time — it turns out that one can do better at ski rental.

§20.1.2 Self-organizing data structure

The idea behind self-organizing data structures is that some items/locations are more expensive to access than others. The data structure can strategically adjust which items are stored where, with the goal of keeping the total cost of queries cheap compared to the optimum, for all access sequences. Intuitively, we're usually willing to put less frequently accessed data in the harder-to-reach places for the sake of keeping our favorite data close.

Example 20.4 (Linked list)
Consider the linked list data structure, which holds n items and such that accessing the item in location j costs j. How should it rearrange its entries as it processes a sequence of queries?

Analysis. Of course, if we knew all frequency counts ahead of time and we wanted a static solution, we could sort the data by frequency. This would also be optimal in a model where requests are generated independently at random from a known distribution.

Here are some natural heuristics:

1. Static: don't rearrange the items after arranging them once.
2. Frequency count (FC): constantly keep the elements sorted by how often they have been accessed over the full history of queries.
3. Move to front (MTF): after each query, move the most recently accessed element to front of the list.
4. Transposition (TRANS): after each query, move the most recently accessed element one position forward.

Historically, probabilistic analysis first suggested that the expected cost of TRANS is less than that of MTF. This disagreed starkly with observations on real data, though, leading to the development of competitive analysis.

Now let's think about the bad inputs for each of these heuristics:

1. Static: the adversary keeps accessing the last element under whichever arrangement we chose. Then each query costs n per access, and would have cost 1 per access in an offline solution.
2. TRANS: the adversary keeps accessing the last element repeatedly. Then the last two entries of the list will keep swapping positions. The total cost of these queries is n per access, though it could have been alternating costs of 1 and 2 in an offline solution.
3. FC: one possibility is to have queries that just cycle through all the elements, so that the empirical distribution doesn't help you. But then the optimal solution also suffers high cost, so this doesn't prove much about FC. We'll pick up here next time with a more clever adversary for frequency count.

□

Remark 20.5. The last point reveals a powerful technique for algorithm design in online learning, and even more generally. A guiding principle in algorithm design is not just to produce good outputs but also to cover your behind, so to speak. Namely, make sure that any bad input for your algorithm needs to be so perverse that the optimal solution also

suffers. So an adversary can't hurt you too much without hurting itself.

§21 Wednesday, November 2

§21.1 Online algorithms II

§21.1.1 Self-organizing data structures II

Let's keep analyzing the frequency count (FC) heuristic for the self-organizing linked list, and work out a lower bound for its competitive ratio. The sequence of operations we'll consider is as follows: access the first element k many times, then the second element $k-1$ times, then the third element $k-2$ times, and so on. Our FC protocol will never rearrange the linked list when fed these queries, so the total cost is

$$\begin{aligned}\sum_{i=1}^{n}(k-i)\cdot i &\geq \sum_{i=n/2}^{n}(k-n)\cdot n/2\\ &\geq \frac{n^2}{4}\cdot(k-n).\end{aligned}$$

The OPT, meanwhile, could move each element to the first place after the first access, so

$$\begin{aligned}\text{cost}(\mathsf{OPT}) &\leq n\cdot n+1\cdot\sum_{i=1}^{n}(k-i)\\ &\leq n^2+kn.\end{aligned}$$

Then our competitive ratio is at least

$$\frac{\frac{n^2}{4}(k-n)}{n^2+k\cdot n}\longrightarrow\frac{n}{4}.$$

Remark 21.1. The intuition here is that frequency count doesn't respond to changes in the query structure fast enough.

Theorem 21.2

Move-to-front (MTF) is 2-competitive. Namely, for all sequences σ, $\text{cost}(MTF(\sigma))\leq 2\cdot\text{cost}(\mathsf{OPT}(\sigma))$.

Proof. Ideally, we would like to prove that the inequality holds in each iteration. This will not be true: OPT could after iteration 1 pay $n-1$ to move the last element to first place. Then, in iteration 2, OPT will pay 1 for its access while MTF will pay n.

To deal with this, we'll amortize the costs of operations. The key observation is as follows: if the state of the linked lists of MTF and OPT are "similar" after $i-1$ iterations, then the access costs for iteration i should also be "similar".

We'll define a potential function $\phi(i)$ that measures how similar the linked lists of ϕ and OPT are at step i. Then we'll define a modified cost for MTF of the form:

$$\text{cost}'(MTF(i)):=\text{cost}(MTF(i))+\phi(i)-\phi(i-1).$$

And the total modified cost is an upper bound on the sum of actual costs, as the $\phi(i)$ telescope, leaving one with $\phi(\ell)-\phi(0)=\phi(\ell)\geq 0$. Now we'll show that for all i, $\text{cost}'(MTF(i))\leq 2\cdot\text{cost}(\mathsf{OPT}(i))$, at which point we're done.

So what should ϕ be? There are two natural choices:

1. Spearman's footrule: $\sum_j |p_{\mathsf{OPT}}(j) - p_{MTF}(j)|$, where $p(j)$ represents the position of element j.
2. Kendall's τ: number of pairs (j, j') with $p_{MTF}(j) < p_{\mathsf{OPT}}(j')$ and $p_{\mathsf{OPT}}(j) > p_{MTF}(j')$. (I.e., number of inversions.)

It turns out that (1) won't work, but (2) will work. The analysis isn't terribly complicated, but I opted to try to follow it carefully rather than write it down. □

Remark 21.3. This analysis relies crucially on the fact that all transpositions have cost 1 in this model (and these are the only swaps allowed, other than move-to-front). That's not too realistic for linked lists, but it's needed to make this analysis go through.

§21.2 Randomized algorithms

Definition 21.4 — A **randomized algorithm** is one that is allowed to use random bits when making decisions. The are essentially two kinds:

- **Monte Carlo algorithms**, whose *output* can vary with respect to the random bits, and
- **Las Vegas algorithms**, whose *runtime* and even termination can vary with respect to the random bits.

A large unsolved question is whether RP = P, i.e., whether there are any problems that can only be solved in polynomial time by randomized algorithms.

Example 21.5 (Max-3-SAT)
Given a 3-SAT formula with n variables and m clauses, find an assignment maximizing the number of satisfied clauses.

Solution. We'll use **Johnson's algorithm**, which is defined as simply choosing a uniformly random variable assignment (without even looking at the clauses!). This begets a random variable Z for the number of satisfied clauses. To compute $\mathbb{E}\,Z$ — and get a handle on the performance of this algorithm — we write $Z = \sum_j Z_j$, where Z_j records whether clause j is satisfied or not. We have probability at least $7/8$ of satisfying each clause, so $\mathbb{E}\,Z \geq \frac{7}{8} \cdot m$ and we have a $7/8$-approximation. This also implies that one can always satisfy at least a $7/8$ fraction of a 3-SAT's clauses using some variable assignment, which is not obvious a priori. □

§22 Monday, November 7

§22.1 Randomized algorithms II

Last time we learned about Johnson's algorithm as a randomized approximation to Max-3-SAT. It attains a respectable approximation factor of $7/8$, but it also seemed pretty silly — it doesn't even require that we look at the clauses! *Obviously*, we can get a better approximation by putting in some more thought. Or can we...

Theorem 22.1 (Hastad)
Unless P = NP, there is no poly-time $7/8+\epsilon$ approximation algorithm for MAX-3-SAT for any constant $\epsilon > 0$.

We starting class by walking through the familiar coupon collector problem. One remarkable fact is that if you only need to see 90% of the coupons, then your average waiting time goes down from $n\ln(n)$ to $n\ln(10)$. Real world takeaways:

- If you've already collected 17 of the 20 coupons you need, you're actually not close to being finished at all! It's the last couple where the waiting time really kicks in.
- If you can design some process (e.g., in machine learning or communication) so that you only need to see 95% of the data to succeed, rather than 100%, you've improved things tremendously!

§22.2 Median finding

Problem 22.2 (Median finding)**.** Given an array of integers, find its median.

There are three ways to tackle this.

1. Sort the array and index into its median. This is in $\Theta(n\log n)$, which is a bit overkill.
2. There's a deterministic $\Theta(n)$ solution, known as the median of medians.
   - It's quite beautiful, but beyond the scope of our current discussion.
3. Randomized: pick a uniformly random element of the array as a pivot. Divide elements (in $\Theta(n)$) into those smaller than the pivot and those larger. Search in the correct subset.

With the third strategy, we end up with an expected runtime of $T(n) \le \Theta(n) + T(\max(n^-, n^+))$, where n^-, n^+ are the sizes of the sets of elements smaller than our pivot and larger than our pivot, respectively. If we were magically guaranteed that $n/4 \le n^-$ and $n^+ \le 3n/4$, then we would have an expected runtime of

$$T(n) \le \Theta(n) + T(3n/4) \le 4\Theta(n) = \Theta(n).$$

So we would be in business if this were true. How strong is this assumption? Well, our pivot element p has probability $1/2$ of being in the middle 50% of elements in the array, so our previous condition has probability $1/2$ of holding.

Now let X_i be the number of rounds from when our set had size $\le (\frac{3}{4})^{i-1}\cdot n$ for the first time until it had size $(\frac{3}{4})^i \cdot n$ for the first time. Then our total runtime is bounded by $\sum_{i=1}^{\infty} X_i \cdot (\frac{3}{4})^{i-1}\cdot n$. In expectation, this is then bounded by

$$n\sum_{i=1}^{\infty}\left(\frac{3}{4}\right)^{i-1}\mathbb{E}\,X_i \le 2n\cdot\sum_{i=1}^{\infty}\left(\frac{3}{4}\right)^{i} = 8n = \Theta(n).$$

§22.2.1 Quicksort

The randomized sorting algorithm **quicksort** is defined as follows: pick a pivot p uniformly at random and proceed essentially as in our median finding algorithm. To analyze the number of comparisons demanded by quicksort, let the random variable $X_{i,j}$ be 1 if elements i and j are ever compared by the algorithm and 0 otherwise. The key observations are as follows:

1. If elements i, j are split into separate groups before one becomes a pivot, then they will never be compared.
2. If i or j is selected as a pivot before they are split into separate groups, then they will be compared.

When a pivot point is drawn that is less than i or greater than j — let's assume $i < j$ — then nothing happens. So we can fast forward to the first time a pivot p is drawn form the set $\{i, i+1, \ldots, j\}$. At that point, the probability of comparing them is exactly $\frac{2}{j+1-i}$, corresponding to the event $p \in \{i, j\}$. So we have

$$\begin{aligned}
\mathbb{E}(\#\text{ of comparisons}) &= \mathbb{E}\left(\sum_{i,j} X_{ij}\right) \\
&= \sum_{i,j} \mathbb{E}\, X_{ij} \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j+1-i} \\
&= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \\
&\leq 2 \cdot \sum_{i=1}^{n} \sum_{k=1}^{n} \frac{1}{k} \\
&\approx 2n \ln(n).
\end{aligned}$$

Remark 22.3. Fastforwarding until a pivot was selected in the interval $\{i, \ldots, j\}$ was a key move in the previous result. It's a neat idea.

§22.3 Concentration bounds

Thus far, we've focused on the expectations of the random variables we've encountered. It can also be crucial to have bounds on the probability that a random variable stays close to its expectation (or, relatedly, does not get too large/small). These are known as **concentration bounds**, and one of the most famous is Markov's inequality. It is very broadly applicable, but sometimes too weak to be of much use.

Theorem 22.4 (Markov's inequality)
If X is a non-negative random variable with a first moment, then $\mathbb{P}(X > a) \leq \frac{\mathbb{E}\, X}{a}$.

Remark 22.5. One situation in which Markov's inequality is useful is that in which you have a sequence of random variables X_i with $\mathbb{E}(X_i) \to 0$ very quickly.

When working with a sum of several independent random variables rather than a single one, the stronger Chernoff bounds can be useful.

Theorem 22.6 (Chernoff bounds)
Let $X_1, \dots, X_n$ be independent 0-1 random variables, $X = \sum_{i=1}^n X_i$, and $\delta > 0$. Then,

1. $\mathbb{P}(X > (1+\delta)\mu) < \left(\frac{e^\delta}{(1+\delta)^{1+\delta}}\right)^\mu$, for any $\mu \geq \mathbb{E}\, X$,
2. $\mathbb{P}(X < (1-\delta)\mu) < e^{-\frac{1}{2}\mu\delta^2}$, for any $\mu \leq \mathbb{E}\, X$.

§23 Wednesday, November 9

§23.1 Concentration bounds II

Let's continue our discussion of concentration bounds with the Hoeffding bound.

Theorem 23.1 (Hoeffding)
Let $X_1, \dots, X_n$ be independent random variables with $X_i \in [a_i, b_i]$ and $X = \sum_i X_i$. Then,

$$\mathbb{P}(|X - \mathbb{E}\, X| \geq t) \leq 2e^{-\frac{2t^2}{\sum_i (b_i - a_i)^2}}.$$

Example 23.2 (Load Balancing)
We have m identical jobs and n machines. The goal is to distribute jobs across machines without coordination. That is, we don't want the hassle of making machines speak with each other. One approach is to have each job assigned independently and uniformly at random to a machine. What is the maximum load on any machine?

Solution. To get a bound on the probability that the load for a *given* machine grows too large, we can use the Chernoff bound. Let's pick δ so that our probability guarantee is $< \frac{1}{n^2}$. In particular, this will allow us to take a union bound over our n machines while still keeping the probability fairly small.

As the average for our given machine's load X is m/n, the Chernoff bound gives us for any $\delta > 0$,

$$\mathbb{P}\left(X > (1+\delta)\cdot\frac{m}{n}\right) < \left(\frac{e^\delta}{(1+\delta)^{1+\delta}}\right)^{\frac{m}{n}}.$$

So let's engineer this to get a probability guarantee of $\frac{1}{n^2}$. Unfortunately, our left hand side has copies of δ in the numerator and denominator, and these kinds of expressions usually can't be solved in closed form.

$$\left(\frac{e^\delta}{(1+\delta)^{1+\delta}}\right)^{\frac{m}{n}} < \frac{1}{n^2}$$

This is a bit of a hassle to solve, but a key point is that there are two regimes here:

1. $m = n$ (or $m \approx n$): Then we need

$$\frac{e^\delta}{(1+\delta)^{1+\delta}} < \frac{1}{n^2}.$$

And it turns out that $\delta = \Theta\left(\frac{\log n}{\log \log n}\right)$ is sufficient. This is also pretty tight; you'll see loads about this big — i.e., $(1+\delta)\cdot\frac{m}{n}$ — in practice.

2. $m >> n$: let's assume $m \geq 16 \cdot n \log n$. Then,

$$\left(\frac{e^\delta}{(1+\delta)^{1+\delta}}\right)^{\frac{m}{n}} \leq \left(\frac{e^\delta}{(1+\delta)^{1+\delta}}\right)^{16\log n}$$

as the term being exponentiated is ≤ 1. Setting $\delta = 1$, the right hand side comes out to:

$$\left(\frac{e}{4}\right)^{16\log n} \leq \left(\frac{1}{e^2}\right)^{\log n} = \frac{1}{n^2}.$$

So in this regime, $\delta = 1$ is sufficient.

By the union bound over all machines, with probability at least $1 - \frac{1}{n}$, all loads are bounded by $(1+\delta)\cdot\frac{m}{n}$. □

Remark 23.3. For all $x \geq 0$, it's true that

$$(1+x) \leq e^x \leq (1+x)^{1+x}.$$

§23.1.1 Route selection

Problem 23.4. We are given an undirected graph G, with pairs of vertices (s_i, t_i). The goal is to pick a path $P_i : s_i \to t_i$ for each i, minimizing the maximum edge congestion (i.e., maximum number of paths selecting any one edge).

If we ask for congestion 1, this is edge-disjoint paths, which is NP-hard for directed graphs even for 2 pairs, and for undirected graphs when the number of pairs is part of the input. So we'll need an approximation algorithm.

[We spent the rest of the lecture devising and analyzing an approximation algorithm. It was pretty involved, and I opted to pay full attention rather than transcribe.]

§24 Monday, November 14

§24.1 Packet routing in networks

Problem 24.1 (Packet routing)**.** We are given a graph G with N paths $P_1, \ldots, P_N$. For each path, there is a packet that needs to traverse that path. For each edge e in G, only one packet can cross e per time step (others can wait in queue). Now the goal is to finish routing packets in as few rounds as possible.

Before even thinking about an algorithm here, let's derive some lower bounds.

- dilation $d := \max_i |P_i|$
- congestion $c = \max_e \left|\{i : e \in P_i\}\right|$

And an easy upper bound on the optimum here is $c \cdot d$: each step on each path has to wait at most c time steps to cross each edge. (So long as we don't leave edges unused when there's demand for them.)

Remark 24.2. It's actually possible to show that $O(c+d)$ is possible, though we won't discuss that here.

We'll take a randomized approach. It's one of those tactics that can be difficult to motivate if you haven't seen it, but feels fairly natural once you've used it a few times.

Algorithm 24.3 — Each packet independently draws a uniformly random start time $s_i \in \{1, \dots, r\}$, for some r to be determined. Then, it tries to follow one edge in each "time step."

Proof sketch. We'll choose r so that in each time step t, at most a small number b of packets want to cross any one edge e. Then, by replacing each time step with a "meta time step" of b real time steps, we slow down the process by a factor b and ensure that all packets can progress through their edges in one meta time step. The total time required will then be $O\big((r+d)\cdot b\big)$. □

Proof. We reverse engineer r, b to make this work. Consider an edge e in time step t. Set $X_{e,t}$ to be the number of paths crossing e at step t. We'll be assuming for now that packets can cross edges at the same time. So $\mathbb{E}(X_{e,t}) \le \frac{c}{r}$. In particular, each of the $\le c$ paths hitting e have probability $\frac{1}{r}$ of having started at exactly the right time so that they're crossing edge e at time t.

So we need r, b such that $\mathbb{P}(X_{e,t} > b) < \epsilon$ for $\epsilon \le \frac{1}{m \times m \cdot (r+d)}$. Then we can apply a union bound over our $m \cdot (r+d)$ pairs of (edge, time step). And because packet starting times are independent, the $X_{e,t,i}$ are independent for fixed e, t and varying packets $i \in [N]$. Thus we can apply the Chernoff bound.

We use $\mu = c/r \ge \mathbb{E}(X_{e,t})$ and somewhat arbitrarily choose $\delta = 1$. Then we have

$$\mathbb{P}\Big(X_{e,t} \ge 2 \cdot \frac{c}{r}\Big) < \Big(\frac{e}{4}\Big)^{c/r}.$$

So we solve $\left(\frac{e}{4}\right)^{c/r} < \frac{1}{m^2(r+d)}$. This gives us

$$\begin{aligned}\Big(\frac{e}{4}\Big)^{c/r} &< \frac{1}{m^2(r+d)} \\ c/r &> \log_{4/e} m^2(r+d) \\ r &< \frac{c}{\log_{4/e} m^2(r+d)}.\end{aligned}$$

And because $r \le c \le N$, and $d \le m$, we get that

$$r < \frac{c}{\log(m^2(N+M))}$$

is sufficient. This puts r in $\Theta(\frac{c}{\log(mN)})$. So we pick r like so, our union bound works, and we get $b = 2\frac{c}{r} = \Theta(\log mN)$. Then the total number of real time steps is

$$\begin{aligned}b \cdot (d+r) &= \Theta(\log(mN)) \cdot \left(d + \frac{c}{\Theta(\log MN)}\right) \\ &= \Theta\Big(c + d\log(mN)\Big),\end{aligned}$$

and this holds with probability $\ge 1 - \frac{1}{m}$. □

§24.2 Game theory

Definition 24.4 — **Game theory** is an analysis framework for situations in which $n \geq 2$ players each take actions, and the actions taken by all players jointly determine an outcome, which in turn determines each player's utility.

The simplest case here is two-player games, $n = 2$. In this case, if each player chooses just one action, we can encode outcomes using matrices. Namely, let $A = (A_{r,c})$ equal the reward to the row player if she takes action r and the column player takes action c. Likewise for $B = (B_{r,c})$ and the utility to the column player. In **zero-sum games**, $B = -A$, i.e., one player's gain is the other's loss.

Remark 24.5. A zero-sum two-player game can be thought of as follows. Players A and B each take actions, and at the conclusion of the turn one player pays the other k many dollars.

Lemma 24.6

Going first in a zero-sum game is never better than going second.

Proof. Say we're the row player. If we go first against an optimal adversary, we'll receive utility $\max_r \min_c A_{r,c}$. If we go second, we get $\min_c \max_r A_{r,c}$. We have:

$$\max_r \min_c A_{r,c} = \min_c A_{\hat{r},c} \leq A_{\hat{r},\hat{c}} \leq \max_r A_{r,\hat{c}} = \min_c \max_r A_{r,c}.$$

□

§25 Wednesday, November 16

§25.1 Game theory II

Definition 25.1 — A **pure Nash equilibrium** in a game is an assignment of actions to players such that no player benefits from changing their strategy in isolation.

Remark 25.2. We're using *strategy* here to simply mean action. Later on, we'll slightly generalize this.

A pure Nash equilibrium is a reasonable and stable outcome in that — supposing the players ever reach it — no one will ever want to deviate unilaterally. Notably, we're not considering chain effects or cooperation here (e.g., player A subsequently changing their strategy in response to the change from player B, or players A and B coordinating ahead of time to both change their actions simultaneously).

Definition 25.3 — A **dominant strategy** is one that is optimal regardless of what other players are doing. (It frequently does not exist.)

In the tragedy of the commons, the key issue that selfishness is a dominant strategy. Regardless of how much the other people in your community are fishing in the lake, you're better off fishing more than your fair share. Likewise, ratting out your friend is

the dominant strategy in the prisoner's dilemma. So these are settings where dominant strategies exist and are a really bad thing — they hurt everyone. Note also that pure Nash equilibria may not exist; consider the (zero-sum!) game of Rock, Paper, Scissors (RPS).

In RPS, you can get away from this using randomization. For instance, if you're going first in RPS, the best you can do is to say that you'll play each move with probability 1/3.

Definition 25.4 — A **mixed Nash equilibrium** is an assignment of distributions over actions such that no player wants to deviate from their strategy in isolation.

Theorem 25.5 (Nash)

Any finite bimatrix game[a] has a mixed equilibrium.

[a] i.e., two-player simultaneous game with a fixed number of actions.

Proof. Non-constructive, using a fixed-point argument. More recent proofs have shown that learning algorithms for these games will converge to a mixed equilibrium. □

Actually finding mixed Nash equilibria (NE) is computationally intractable, though.

Theorem 25.6 (DGP)

Finding a mixed NE for $n \geq 3$ players is PPAD-complete.

Remark 25.7. If you're doing research and you find that the notion of mixed Nash equilibrium comes up — or even just of mixed strategies — always make sure to check that it even makes sense to discuss mixed strategies in this setting. For instance, it often doesn't make much sense to discuss a mixed strategy if you only play the game once.

Hereafter we'll be restricting focus to zero-sum games. For two players, one neat observation is that even if the first player's strategy is mixed, the second player's optimal strategy can be assumed to be deterministic, by linearity of expectation.

So let the payoff matrix be $A_{r,c}$ from the perspective of the row player, and suppose that the row player goes first, with mixed strategy $\{p_r\}_{r \in R}$, $\sum_r p_r = 1$. Then the column player will play some deterministic action c, giving the row player a payoff of $\sum_r p_r A_{r,c}$. In particular, the column player will select the action c to *minimize* this sum, as we're in a zero-sum game. So the goal of the row player is to select p_r so as to maximize $\min_c p_r A_{r,c}$.

This can be phrased as a linear program, as follows:

$$\begin{aligned} &\text{maximize } W \\ &\text{such that } W \leq \sum_r p_r \cdot A_{r,c} \ \forall c \\ &\sum_r p_r = 1 \\ &p_r \geq 0 \ \forall r \end{aligned}$$

Any time we have a linear program, we should take its dual and see what it tells us. In this case — with a bit of manual labor — we can see that the dual captures the column

player's optimization problem for finding an optimal first-mover strategy (i.e., if the column player were to move first). From weak duality, we then have the following result.

Theorem 25.8 (Loomis's Theorem)
For all distributions P, Q,

$$\min_c \mathop{\mathbb{E}}_{r\sim P} A_{r,c} \le \max_r \mathop{\mathbb{E}}_{c\sim Q} A_{r,c}.$$

And from strong duality, we have that if P and Q are *optimal* strategies for the row and column players, then

$$\min_c \mathop{\mathbb{E}}_{r\sim P} A_{r,c} = \max_r \mathop{\mathbb{E}}_{c\sim Q} A_{r,c}.$$

So if both players play optimally, then the order doesn't matter, and (P, Q) is a mixed equilibrium. This is known as **von Neumann's minimax theorem**.

Game theory is a beautiful field currently receiving lots of attention in computer science. An important observation is that algorithm design itself can be viewed as a 2-player game between an algorithm designer and an adversarial input designer. In particular, let $\mathcal{A}$ be the set of all deterministic algorithms (say for fixed input size n), and let $\mathcal{I}$ be the set of all inputs of size n. And let the utility to our adversary for (A, I) be the runtime of algorithm A on input I (or memory usage, approximation guarantee, etc.). Then this game precisely captures algorithm design.

We will now consider randomized algorithms A that always terminate within $f_A(n)$ many steps. These can be viewed as a distribution over at most $2^{f_A(n)}$ deterministic algorithms. (We can flip $f_A(n)$ many coins before the algorithm starts running; whenever it needs another coin, it reads the next pre-flipped coin.) Let P be the resulting distribution over deterministic algorithms. Then the worst-case expected performance of P is $\max_{I\in\mathcal{I}} \mathbb{E}_{A\sim P} \operatorname{cost}(A, I)$.

And by Loomis's theorem, we have, for *any* P and Q,

$$\max_{I\in\mathcal{I}} \mathop{\mathbb{E}}_{A\sim P} \operatorname{cost}(A, I) \ge \min_A \mathop{\mathbb{E}}_{I\sim Q} \operatorname{cost}(A, I).$$

This is known as **Yao's minimax principle**.

Theorem 25.9 (Yao's minimax principle)

Fix an input size n, let $\mathcal{A}$ be the set of all correct deterministic algorithms for a given problem with size n, and $\mathcal{I}$ the set of all inputs of size n. Let $\text{cost}(A, I)$ be any cost measure for algorithm $A \in \mathcal{A}$ running on input $I \in \mathcal{I}$ (runtime, memory, approximation factor, etc.).

Now let P be any distribution over algorithms with finite support, and Q any distribution over inputs. Then

$$\min_{A \in \mathcal{A}} \mathop{\mathbb{E}}_{I \sim Q} \text{cost}(A, I) \leq \max_{I \in \mathcal{I}} \mathop{\mathbb{E}}_{A \sim P} \text{cost}(A, I).$$

So the running time of the *best* deterministic algorithm for any input distribution Q is a lower bound for the running time of *any* randomized algorithm[a] for its worst-case input. (Equality follows from von Neumann's theorem if P is the best randomized algorithm and Q is the worst input distribution.)

[a] Strictly speaking, randomized algorithm that uses a *bounded* number of coin flips on inputs of size n.

§26 Monday, November 21

§26.1 AND/OR Tree evaluation

Problem 26.1. We have a tree whose leaves are labeled as True or False, and whose remaining nodes are labeled by AND or OR. An AND node evaluates to True if all its subtrees evaluate to True, and an OR node evaluates to True if any one of its subtrees evaluate to True. The layers of the tree alternate between consisting of all AND nodes and all OR nodes. The goal is to evaluate the root of the tree while querying as few leaves as possible.

Lemma 26.2

Any deterministic algorithm can be forced to evaluate all the leaves.

Proof sketch. Show by induction that for any height h and any type of root, a deterministic algorithm can be forced to evaluate all leaves *and* the outcome can be True or False. This completes the proof. Furthermore, as with other online settings, because the algorithm is deterministic, the adversary can specify the whole input ahead of time. □

Now let's see if randomness can help us. We'll assume that our trees are binary for the sake of simplicity, though it doesn't affect the analysis very much. Let's now design a randomized algorithm in which we process nodes from the top down. For instance, to learn the value of a tree with an AND root:

- If both children are 1(=True), then both have to be evaluated.
- If both children are 0(=False), then only one has to be evaluated.
- If one is 0 and the other is 1, then with the wrong order of evaluation (i.e., evaluating 1 first), both children will have to be evaluated. Otherwise, only one will be. We'll randomize that order of evaluation to get it right with probability 1/2.

Similarly, for an OR node, if it evaluates to 1, then with probability at least 1/2, one only has to evaluate a single subtree.

Algorithm 26.3 (Snir's algorithm) — To evaluate a node, evaluate a uniformly random child, then evaluate the other child if necessary.

You can think of this as a randomized depth-first search. – David

Analysis of Algorithm 26.3. To analyze this, let $X_k^{0/1}$ be the random variable that gives the cost of evaluating a node at level k when the outcome is 0/1. Now let k be even, so that we have an AND node. Then to evaluate it to 1, we needed to have looked at both of its children, so

$$X_k^1 \le 2X_{k-1}^1 \implies \mathbb{E}(X_k^1) \le 2\,\mathbb{E}(X_{k-1}^1).$$

To evaluate it to 0, at least one of its children must evaluate to 0, so

$$X_k^0 \le X_{k-1}^0 + [\text{wrong choice}] \cdot X_{k-1}^1 \implies \mathbb{E}(X_k^0) \le \mathbb{E}(X_{k-1}^0) + \frac{1}{2}\,\mathbb{E}(X_{k-1}^1).$$

Symmetrically, for k odd, giving us an OR node, we have

$$\mathbb{E}(X_k^0) \le 2\,\mathbb{E}(X_{k-1}^0), \qquad \mathbb{E}(X_k^1) \le E(X_{k-1}^1) + \frac{1}{2}\,\mathbb{E}(X_{k-1}^0).$$

Now write $Y_k = \max\big(\mathbb{E}(X_k^0), \mathbb{E}(X_k^1)\big)$. For k even, we have

$$E(X_k^1) \le 2\,\mathbb{E}(X_{k-1}^1) \le 2\,\mathbb{E}(X_{k-2}^1) + \mathbb{E}(X_{k-2}^0) \le 3Y_{k-2}.$$

And likewise,

$$\mathbb{E}(X_k^0) \le \mathbb{E}(X_{k-1}^0) + \frac{1}{2}\,\mathbb{E}(X_{k-1}^1) \le 2\,\mathbb{E}(X_{k-2}^0) + \frac{1}{2}\,\mathbb{E}(X_{k-2}^1) + \frac{1}{4}\,\mathbb{E}(X_{k-2}^0) \le 3Y_{k-2}.$$

Similarly for an OR node (i.e., k odd), we can show that $Y_k \le 3Y_{k-2}$. So a tree with k AND nodes and k OR nodes — that is, height $2k$ — has $Y_{2k} \le 3^k$. And such a tree has $n = 2^{2k} = 4^k$ leaves. So $k = \log_4 n$, and the expected number of evaluations is at most

$$3^{\log_4(n)} = n^{\log_4(3)} \approx n^{0.793}.$$

This also implies the existence of a certificate of size $\le n^{0.793}$ for every possible truth setting, which is not so obvious. □

We'd now like to derive a lower bound for this problem. That seems pretty challenging to do directly, especially as we're analyzing randomized algorithms, and we don't know their coin flips ahead of time. To derive a lower bound, we'll appeal to Yao's minimax principle. We'll need to describe a malicious input distribution, then analyze the best deterministic algorithm against that distribution. Of course, every distribution will give *some* lower bound, but not-so-clever choices — e.g., distributions with small support — will give weak lower bounds.

A natural starting point is to set each leaf to True independently with probability p. Taking $p = \frac{1}{2}$ seems reasonable enough, but it has the disadvantage that two levels up, the probability of a subtree being true is $9/16 \ne 1/2$. That seems somewhat unfortunate for our analysis, so we'll instead choose p such that two levels up in our tree, the probability of a node evaluating to True remains constant. Chasing a bit of algebra, this comes out to demanding that $p = (1-p)^2$, leaving us with $p = \frac{3-\sqrt{5}}{2}$. One can show — though it

takes work — that the optimal deterministic algorithm for this input distribution is to query leaves $1, \dots, n$ in order, skipping leaves whose values are not needed. (And this is equivalent to a deterministic analogue of our previous DFS.)

So what's the expected cost of this algorithm? Let $T(k)$ be the number of queries it makes to evaluate a tree of height k against our input distribution. We have

$$\mathbb{E}(T(k)) \geq \mathbb{E}(T(k)) + (1-p)\,\mathbb{E}(T(k-1)).$$

In particular, we always evaluate one tree, and we evaluate the second if and only if the first one was false.[11] So,

$$\begin{aligned}
\mathbb{E}(T(k)) &\geq (2-p)\,\mathbb{E}(T(k-1)) \\
&\geq (2-p)^k \\
&= \left(\frac{\sqrt{5}+1}{2}\right)^{\log_2(n)} \\
&= n^{\log_2(\sqrt{5}+1)-1} \\
&\approx n^{0.694}.
\end{aligned}$$

That's a pretty good lower bound, but it doesn't quite match our upper bound. We know that Yao's minimax principle is tight for optimal randomized algorithms & randomized inputs, so one of our bounds must be loose.

It's probably the lower bound that is loose. One reason why we should suspect this is that our input distribution technically allows for silly things like leaves that are all True or all False. More generally, due to independence, our distribution can generate inputs on which the algorithm cannot go wrong (e.g., generating a tree with an AND node whose children both evaluate to False, or an OR node whose children both evaluate to True). Instead, one should anti-correlate the values of subtrees, and generate the input distribution itself using a random DFS assignment.

This makes it more complex to analyze the best deterministic algorithm, but you can in fact show that it gives an $n^{\log_4(3)}$ lower bound. So that one's tight!

Remark 26.4. If you find this kind of thing interesting, here are some open directions:

1. What's the best deterministic algorithm when the leaves ℓ_i of the tree are True independently with known probabilities p_i?
2. What's the best randomized algorithm when there are non-uniform query costs?

§27 Monday, November 28

§27.1 Multiplicative weights

Here's the basic setup of **multiplicative weights**: we want to make a sequence of decisions (for now, binary), and we're equipped with the recommendations of n "experts." The setting is online, so after the algorithm makes its irrevocable decision at each time step, the rewards & penalties of all actions are revealed. Notably, this is hopeless in the "approximation" or "competitive analysis" sense; you can easily be wrong every time.

As a result, we need a more meaningful notion of performance. Rather than competing with an offline solution, our goal will be to do not too much worse than the single best

[11] At one point in the analysis, we turned all the nodes into NOR nodes. I didn't write down the details.

expert (judged in hindsight). These experts can be good, bad, or mixed, and the basic idea of our algorithm will be to follow the experts who have done well so far. That way, in order to make the algorithm look bad, an adversary also has to make all experts look bad. Now for the algorithm.

Algorithm 27.1 (Weighted Majority) —

- fixed parameter η
- initialize all weights $w_i^1 = 1$
- in each round t:
  - choose an action based on the weighted majority of experts
  - for every expert who predicted wrong, set $w_i^{t+1} = (1-\eta)w_i^t$

Theorem 27.2
Let $M^{(T)}$ be the number of mistakes committed by Weighted Majority up to time T, and $m_i^{(T)}$ the number of mistakes committed by expert i up to time T. Then,

$$M^{(T)} \leq \frac{2}{1-\eta} m_i^{(T)} + \frac{2}{\eta}\ln(n)$$

for all experts i.

Proof. Let's define $\Phi_t = \sum_i w_i^{(t)}$. Assume that in round t, our algorithm made a mistake. Then the set S_t of experts making a wrong recommendation had at least half the total weight. This is now multiplied by $(1-\eta)$. So,

$$\Phi_{t+1} \leq \frac{1}{2}\Phi_t + \frac{1}{2}\Phi_t(1-\eta) = \Phi_t(1-\frac{\eta}{2}).$$

And thus,

$$\Phi_T \leq (1-\frac{\eta}{2})^{M(T)} \cdot \Phi_1 = n \cdot (1-\frac{\eta}{2})^{M(T)}.$$

Now fix an expert i. We have that $w_i^{(T)} = (1-\eta)^{m_i(T)}$. And obviously $w_i^{(T)} \leq \Phi_T$, so

$$(1-\eta)^{m_i(T)} \leq n \cdot (1-\frac{\eta}{2})^{M(T)}.$$

And that's really the heart of the analysis! The rest of the work will just be in teasing this out and organizing it more nicely. In particular, we have

$$\begin{aligned}
(1-\eta)^{m_i(T)} &\leq n \cdot (1-\frac{\eta}{2})^{M(T)} \\
\left(\frac{2}{2-\eta}\right)^{M(t)} &\leq n \cdot \left(\frac{1}{1-\eta}\right)^{m_i(t)} \\
M^{(T)} \cdot \ln\frac{2}{2-\eta} &\leq \ln(n) + m_i^{(T)} \cdot \ln\frac{1}{1-\eta}.
\end{aligned}$$

Chasing this further, and using the fact that $\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \ldots \in \left[x - \frac{x^2}{2}, x\right]$, we arrive at our equation. □

Remark 27.3. Two analysis facts we made use of in the previous proof:

- truncating the Taylor series for $\ln(1+x)$ after a negative (resp. positive) term yields a lower (resp. upper) bound on the function,
- $\ln(1+x) \le x$.

It's worth making a distinction between this kind of setting, known as **expert learning**, and the setup of **bandit learning**. Namely, in expert learning, you observe the rewards of *all* experts at each time step, whereas in bandit learning you observe only the reward of your *chosen* expert (i.e., the expert you followed in that time step).

Now let's generalize the setting so that actions are no longer binary. We'll now allow for the experts to take distinct actions. In fact, we'll abstract away the actions: in each round, each expert will simply incur a cost $m_i^{(t)} \in [-1,1]$. (We'll think of negative costs as rewards.) Our algorithm will thus choose an expert to follow at each round and receive their same reward.

Note that a deterministic algorithm doesn't stand a chance anymore; in each round, the chosen expert can be the only bad one. So we'll need to randomize.

Algorithm 27.4 (Multiplicative Weights) —

- fixed parameter η (learning rate)
- initialize weights $w_i^{(t)} = 1$
- in each round t:
  - let $\Phi^{(t)} = \sum_i w_i^{(t)}$
  - choose expert i with probability $P_i^{(t)} = w_i^{(t)}/\Phi^{(t)}$.
  - for each i, update $w_i^{(t+1)} = w_i^{(t)} \cdot (1 - \eta \cdot m_i^{(t)})$

Theorem 27.5

If all $m_i^{(t)} \in [-1,1]$, then Multiplicative Weights' expected cost is at most

$$\sum_t m^{(t)} \cdot p^{(t)} \le \sum_t m_i^{(t)} + \frac{\eta}{1-\eta} \sum_t |m_i^{(t)}| + \frac{\ln(n)}{\eta}$$

for all experts i.

Proof. Let's again study the evolution of $\Phi^{(t)}$ and $w_i^{(t)}$, and use that $w_i^{(t)} \le \Phi^{(t)}$. We have:

$$\begin{aligned}
\Phi^{(t+1)} &= \sum_i w_i^{(t+1)} \\
&= \sum_i w_i^{(t)} (1 - \eta \cdot m_i^{(t)}) \\
&= \Phi^{(t)} - \eta \cdot \sum_i \frac{w_i(t)}{\Phi^{(t)}} \cdot \Phi^{(t)} \cdot m_i^{(t)} \\
&= \Phi^{(t)} (1 - \eta \cdot p^{(t)} \cdot m^{(t)}) \\
&\le \Phi^{(t)} e^{-\eta p^{(t)} m^{(t)}}.
\end{aligned}$$

And this can again be chased to arrive at the inequality in question, using things like log inequalities. I didn't write down the details. □

§28 Wednesday, November 30

§28.1 Multiplicative weights II

Let's look at some nice corollaries from our analysis of Multiplicative Weights in Theorem 27.5.

Corollary 28.1

For any fixed distribution $\vec{q}$ over experts, the algorithm has expected cost at most

$$\frac{\ln(n)}{\eta} + \sum_t \left(m^{(t)} + \frac{\eta}{1-\eta}|m^t|\right) \cdot \vec{q}.$$

Here $|m^{(t)}|$ means taking coordinate-wise absolute values of the vector $m^{(t)}$.

Proof. Just a convex combination of Theorem 27.5. I.e., multiply the previous theorem by q_i for each expert i, then add the inequalities all up. □

So this allows us to compete with a fixed *distribution* over experts rather than just a single expert. An optimal opponent would never really want to do that — they would put all their eggs with the best expert — but there are some natural settings in which an adversary plays with distributions over experts. We'll see some shortly.

Corollary 28.2

If we're speaking in terms of rewards rather than costs, the Multiplicative Weights algorithm guarantees an expected reward of at least

$$\sum_t \left(m^{(t)} - \frac{\eta}{1-\eta}|m^{(t)}|\right) \cdot \vec{q} - \frac{\ln(n)}{\eta}.$$

Some applications of Multiplicative Weights include:

- Linear/convex/semi-definite programming: experts are variables, and rewards are determined by constraints.
- Machine learning (e.g., classifiers): experts are features, and rewards are determined by correct/incorrect classification.
- Learning mixed strategies in games: experts are pure strategies, and rewards are determined by opponent actions.
- Greedy Set Cover approximation can be analyzed as a special case with $\eta = 1$.

Let's dig deeper into an example of Multiplicative Weights for machine learning.

Example 28.3

Say we have m data points $\left(\vec{a}_i\right)_{i\in[m]}$ in $\mathbb{R}^n$, each equipped with a binary label $\ell_i \in \{\pm 1\}$. We assume the points are linearly separable via a hyperplane $\vec{x}$ going

through the origin with $x_i \geq 0 \ \forall i$.

Now, the assumption that such a hyperplane exists is equivalent to saying that there is a vector $\vec{x}$ such that $\ell_j = \text{sign}(\vec{a}_j \cdot \vec{x})$ for all j. Because $\text{sign}(-\vec{a}_j \cdot \vec{x}) = -\text{sign}(\vec{a}_j \cdot \vec{x})$, we can assume without loss of generality that all $\ell_j = 1$. (More explicitly, we replace any $(\vec{a}_j, 0)$ pair with $(-\vec{a}_j, 1)$.) Furthermore, because $\text{sign}(\vec{a} \cdot \vec{x}) = \text{sign}(c\vec{a} \cdot \vec{x})$ for all $c \geq 0$, we can divide all $\vec{a}_j$ by $\max_i |a_{j,i}|$ if needed and assume that all $a_{j,i} \in [-1, 1]$. We can normalize x_i in the same way, so that $\sum_i x_i = 1$.

In summary, we have $\vec{a}_j$ with $a_{j,i} \in [-1, 1]$, and we want to find $\vec{x}$ with $\vec{a}_j \cdot \vec{x} \geq 0$, $\sum_i x_i = 1$, and $x_i \geq 0$. That's an LP! Yet another assumption we'll now introduce is the existence of a large-margin classifier, with margin at least ϵ. That is, $\vec{x} \cdot \vec{a}_j \geq \epsilon$ for all j.

We now treat each dimension as an expert, run Multiplicative Weights with these experts, and use the probabilities $p_i^{(t)}$ in each round t as our attempted "solution" for the LP. Here's the key idea: in each iteration, unless $\vec{p}^{(t)}$ already solves the LP, there is a point $\vec{a}_j$ with $\vec{a}_j \cdot \vec{p}^{(t)} < 0$. And $\vec{a}_j \cdot \vec{x} \geq \epsilon$.

So our algorithm is:

- Until $\vec{p}^{(t)}$ solves the LP:
  - Pick a point $\vec{a}_j$ with $\vec{a}_j \cdot \vec{p}^{(t)} < 0$
  - Use $a_{j,i}$ as the reward for each expert i, to update $\vec{p}^{(t+1)}$ according to Multiplicative Weights

By our earlier corollary, the expected reward over rounds $t \in \{1, \ldots, T\}$ is at least $\sum_t \vec{x} \cdot (\vec{a}_{j_t} - \eta|\vec{a}_{j,t}|) - \frac{\ln(n)}{\eta}$. And that's at least $T(\epsilon - \eta) - \frac{\ln(n)}{\eta}$. On the other hand, our expected reward is negative. So we have

$$T < \frac{\ln(n)}{\eta(\epsilon - \eta)}.$$

Setting $\eta = \frac{\epsilon}{2}$, we have that $T < \frac{4\ln(n)}{\epsilon^2}$. So we must get a feasible solution after at most $\frac{4\ln(2)}{\epsilon^2}$ rounds! (In particular, note that our algorithm is completely deterministic; the probability distributions we dealt with are just an analysis tool.)

Index