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 $\text{dist}(s,v) \leq d[v] \leq \text{dist}(s,v)$, 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 $c_e \geq 0$, 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. $\square$

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). $\square$

§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 `inserts` 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.

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

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

[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.

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!).

§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

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

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 $A_1 \cdot A_2 \cdot \dots \cdot A_k$, 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 $(A_1 \cdot \dots \cdot A_i) \cdot (A_{i+1} \cdot \dots \cdot A_k)$. 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 $a_1, \dots, a_n$ 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 $x[1, \dots, i]$ with $y[1, \dots, j]$. 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 $P_1, \dots, P_k$ with flow values $\alpha_1, \dots, \alpha_k \geq 0$ 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.

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

□

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 $|S| \le |N(S)|$ 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 $|X \cap \overline{A}| + |Y \cap A| \le n-1$.

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

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

So we're done. □

Corollary 10.3
Every d-regular bipartite graph $d \ge 1$ 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 $P_1, \dots, P_k$ with associated (integer) values $a_1, a_2, \dots, a_k$ 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 $P_1, \dots, P_k$ 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. $\square$

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
P ⊆ 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
NP ⊆ 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 NP $\subseteq P$? Equivalently, is P = 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 P ≠ 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 $Y \in$ P then $X \in$ P, and if $X \notin$ P then $Y \notin$ 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. $\left(X \leq_p Y\right) \wedge \left(Y \leq_p Z\right) \implies X \leq_p Z$.

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–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 $S_1, \dots, S_m \subseteq U$, determine whether there is a **set cover** of size k, i.e., k subsets $S_{j_1}, \dots, S_{j_k}$ 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. □

Index