15-451 Algorithms
Lectures 11-20

Avrim Blum
Department of Computer Science
Carnegie Mellon University

September 29, 2009

Contents

Lecture 11

Dynamic Programming

11.1 Overview

Dynamic Programming is a powerful technique that allows one to solve many different types of problems in time $O(n^2)$ or $O(n^3)$ for which a naive approach would take exponential time. In this lecture, we discuss this technique, and present a few key examples. Topics in this lecture include:

- The basic idea of Dynamic Programming.
- Example: Longest Common Subsequence.
- Example: Knapsack.
- Example: Matrix-chain multiplication.

11.2 Introduction

Dynamic Programming is a powerful technique that can be used to solve many problems in time $O(n^2)$ or $O(n^3)$ for which a naive approach would take exponential time. (Usually to get running time below that—if it is possible—one would need to add other ideas as well.) Dynamic Programming is a general approach to solving problems, much like "divide-and-conquer" is a general method, except that unlike divide-and-conquer, the subproblems will typically overlap. This lecture we will present two ways of thinking about Dynamic Programming as well as a few examples.

There are several ways of thinking about the basic idea.

Basic Idea (version 1): What we want to do is take our problem and somehow break it down into a reasonable number of subproblems (where "reasonable" might be something like n^2) in such a way that we can use optimal solutions to the smaller subproblems to give us optimal solutions to the larger ones. Unlike divide-and-conquer (as in mergesort or quicksort) it is OK if our subproblems overlap, so long as there are not too many of them.

11.3 Example 1: Longest Common Subsequence

Definition 11.1 *The* **Longest Common Subsequence (LCS)** *problem is as follows. We are given two strings: string S of length n, and string T of length m. Our goal is to produce their longest common subsequence: the longest sequence of characters that appear left-to-right (but not necessarily in a contiguous block) in both strings.*

For example, consider:

$$S = \texttt{ABAZDC}$$
$$T = \texttt{BACBAD}$$

In this case, the LCS has length 4 and is the string `ABAD`. Another way to look at it is we are finding a 1-1 matching between some of the letters in S and some of the letters in T such that none of the edges in the matching cross each other.

For instance, this type of problem comes up all the time in genomics: given two DNA fragments, the LCS gives information about what they have in common and the best way to line them up.

Let's now solve the LCS problem using Dynamic Programming. As subproblems we will look at the LCS of a prefix of S and a prefix of T, running over all pairs of prefixes. For simplicity, let's worry first about finding the *length* of the LCS and then we can modify the algorithm to produce the actual sequence itself.

So, here is the question: say `LCS[i,j]` is the length of the LCS of S`[1..i]` with T`[1..j]`. How can we solve for `LCS[i,j]` in terms of the LCS's of the smaller problems?

Case 1: what if $S[i] \neq T[j]$? Then, the desired subsequence has to ignore one of $S[i]$ or $T[j]$ so we have:

$$\boxed{\texttt{LCS}[\texttt{i},\texttt{j}] = \texttt{max}(\texttt{LCS}[\texttt{i}-1,\texttt{j}], \texttt{LCS}[\texttt{i},\texttt{j}-1]).}$$

Case 2: what if $S[i] = T[j]$? Then the LCS of $S[1..i]$ and $T[1..j]$ might as well match them up. For instance, if I gave you a common subsequence that matched $S[i]$ to an earlier location in T, for instance, you could always match it to $T[j]$ instead. So, in this case we have:

$$\boxed{\texttt{LCS}[\texttt{i},\texttt{j}] = 1 + \texttt{LCS}[\texttt{i}-1,\texttt{j}-1].}$$

So, we can just do two loops (over values of i and j) , filling in the LCS using these rules. Here's what it looks like pictorially for the example above, with S along the leftmost column and T along the top row.

	B	A	C	B	A	D
A	0	1	1	1	1	1
B	1	1	1	2	2	2
A	1	2	2	2	3	3
Z	1	2	2	2	3	3
D	1	2	2	2	3	4
C	1	2	3	3	3	4

We just fill out this matrix row by row, doing constant amount of work per entry, so this takes $O(mn)$ time overall. The final answer (the length of the LCS of S and T) is in the lower-right corner.

How can we now find the sequence? To find the sequence, we just walk backwards through matrix starting the lower-right corner. If either the cell directly above or directly to the right contains a value equal to the value in the current cell, then move to that cell (if both to, then chose either one). If both such cells have values strictly less than the value in the current cell, then move diagonally up-left (this corresponts to applying Case 2), and output the associated character. This will output the characters in the LCS in reverse order. For instance, running on the matrix above, this outputs `DABA`.

11.4 More on the basic idea, and Example 1 revisited

We have been looking at what is called "bottom-up Dynamic Programming". Here is another way of thinking about Dynamic Programming, that also leads to basically the same algorithm, but viewed from the other direction. Sometimes this is called "top-down Dynamic Programming".

Basic Idea (version 2): Suppose you have a recursive algorithm for some problem that gives you a really bad recurrence like $T(n) = 2T(n-1)+n$. However, suppose that many of the subproblems you reach as you go down the recursion tree are the *same*. Then you can hope to get a big savings if you store your computations so that you only compute each *different* subproblem once. You can store these solutions in an array or hash table. This view of Dynamic Programming is often called *memoizing*.

For example, for the LCS problem, using our analysis we had at the beginning we might have produced the following exponential-time recursive program (arrays start at 1):

```
LCS(S,n,T,m)
{
  if (n==0 || m==0) return 0;
  if (S[n] == T[m]) result = 1 + LCS(S,n-1,T,m-1); // no harm in matching up
  else result = max( LCS(S,n-1,T,m), LCS(S,n,T,m-1) );
  return result;
}
```

This algorithm runs in exponential time. In fact, if `S` and `T` use completely disjoint sets of characters (so that we never have `S[n]==T[m]`) then the number of times that `LCS(S,1,T,1)` is recursively called equals $\binom{n+m-2}{m-1}$.[1] In the memoized version, we store results in a matrix so that any given set of arguments to `LCS` only produces new work (new recursive calls) once. The memoized version begins by initializing `arr[i][j]` to `unknown` for all `i,j`, and then proceeds as follows:

```
LCS(S,n,T,m)
{
  if (n==0 || m==0) return 0;
  if (arr[n][m] != unknown) return arr[n][m];  // <- added this line (*)
  if (S[n] == T[m]) result = 1 + LCS(S,n-1,T,m-1);
  else result = max( LCS(S,n-1,T,m), LCS(S,n,T,m-1) );
  arr[n][m] = result;                          // <- and this line (**)
```

[1]This is the number of different "monotone walks" between the upper-left and lower-right corners of an n by m grid.

```
  return result;
}
```

All we have done is saved our work in line (**) and made sure that we only embark on new recursive calls if we haven't already computed the answer in line (*).

In this memoized version, our running time is now just $O(mn)$. One easy way to see this is as follows. First, notice that we reach line (**) at most mn times (at most once for any given value of the parameters). This means we make at most $2mn$ recursive calls total (at most two calls for each time we reach that line). Any given call of `LCS` involves only $O(1)$ work (performing some equality checks and taking a max or adding 1), so overall the total running time is $O(mn)$.

Comparing bottom-up and top-down dynamic programming, both do almost the same work. The top-down (memoized) version pays a penalty in recursion overhead, but can potentially be faster than the bottom-up version in situations where some of the subproblems never get examined at all. These differences, however, are minor: you should use whichever version is easiest and most intuitive for you for the given problem at hand.

More about LCS: Discussion and Extensions. An equivalent problem to LCS is the "minimum edit distance" problem, where the legal operations are insert and delete. (E.g., the unix "diff" command, where S and T are files, and the elements of S and T are lines of text). The minimum edit distance to transform S into T is achieved by doing $|S| - \texttt{LCS}(S,T)$ deletes and $|T| - \texttt{LCS}(S,T)$ inserts.

In computational biology applications, often one has a more general notion of sequence alignment. Many of these different problems all allow for basically the same kind of Dynamic Programming solution.

11.5 Example #2: The Knapsack Problem

Imagine you have a homework assignment with different parts labeled A through G. Each part has a "value" (in points) and a "size" (time in hours to complete). For example, say the values and times for our assignment are:

	A	B	C	D	E	F	G
value	7	9	5	12	14	6	12
time	3	4	2	6	7	3	5

Say you have a total of 15 hours: which parts should you do? If there was partial credit that was proportional to the amount of work done (e.g., one hour spent on problem C earns you 2.5 points) then the best approach is to work on problems in order of points/hour (a greedy strategy). But, what if there is no partial credit? In that case, which parts should you do, and what is the best total value possible?[2]

The above is an instance of the *knapsack problem*, formally defined as follows:

[2]Answer: In this case, the optimal strategy is to do parts A, B, F, and G for a total of 34 points. Notice that this doesn't include doing part C which has the most points/hour!

Definition 11.2 *In the* **knapsack problem** *we are given a set of n items, where each item i is specified by a size s_i and a value v_i. We are also given a size bound S (the size of our knapsack). The goal is to find the subset of items of maximum total value such that sum of their sizes is at most S (they all fit into the knapsack).*

We can solve the knapsack problem in exponential time by trying all possible subsets. With Dynamic Programming, we can reduce this to time $O(nS)$.

Let's do this top down by starting with a simple recursive solution and then trying to memoize it. Let's start by just computing the best possible *total value*, and we afterwards can see how to actually extract the items needed.

```
// Recursive algorithm: either we use the last element or we don't.
Value(n,S)    // S = space left, n = # items still to choose from
{
  if (n == 0) return 0;
  if (s_n > S) result = Value(n-1,S); // can't use nth item
  else result = max{v_n + Value(n-1, S-s_n), Value(n-1, S)};
  return result;
}
```

Right now, this takes exponential time. But, notice that there are only $O(nS)$ *different* pairs of values the arguments can possibly take on, so this is perfect for memoizing. As with the LCS problem, let us initialize a 2-d array `arr[i][j]` to "unknown" for all `i,j`.

```
Value(n,S)
{
  if (n == 0) return 0;
  if (arr[n][S] != unknown) return arr[n][S];   // <- added this
  if (s_n > S) result = Value(n-1,S);
  else result = max{v_n + Value(n-1, S-s_n), Value(n-1, S)};
  arr[n][S] = result;                           // <- and this
  return result;
}
```

Since any given pair of arguments to Value can pass through the array check only once, and in doing so produces at most two recursive calls, we have at most $2n(S+1)$ recursive calls total, and the total time is $O(nS)$.

So far we have only discussed computing the *value* of the optimal solution. How can we get the items? As usual for Dynamic Programming, we can do this by just working backwards: if `arr[n][S]` = `arr[n-1][S]` then we *didn't* use the nth item so we just recursively work backwards from `arr[n-1][S]`. Otherwise, we *did* use that item, so we just output the nth item and recursively work backwards from `arr[n-1][S-s_n]`. One can also do bottom-up Dynamic Programming.

11.6 Example #3: Matrix product parenthesization

Our final example for Dynamic Programming is the *matrix product parenthesization* problem.

Say we want to multiply three matrices X, Y, and Z. We could do it like $(XY)Z$ or like $X(YZ)$. Which way we do the multiplication doesn't affect the final outcome but it *can* affect the running time to compute it. For example, say X is 100x20, Y is 20x100, and Z is 100x20. So, the end result will be a 100x20 matrix. If we multiply using the usual algorithm, then to multiply an ℓxm matrix by an mxn matrix takes time $O(\ell mn)$. So in this case, which is better, doing $(XY)Z$ or $X(YZ)$?

Answer: $X(YZ)$ is better because computing YZ takes 20x100x20 steps, producing a 20x20 matrix, and then multiplying this by X takes another 20x100x20 steps, for a total of 2x20x100x20. But, doing it the other way takes 100x20x100 steps to compute XY, and then multplying this with Z takes another 100x20x100 steps, so overall this way takes 5 times longer. More generally, what if we want to multiply a series of n matrices?

Definition 11.3 *The* **Matrix Product Parenthesization** *problem is as follows. Suppose we need to multiply a series of matrices: $A_1 \times A_2 \times A_3 \times \ldots \times A_n$. Given the dimensions of these matrices, what is the best way to parenthesize them, assuming for simplicity that standard matrix multiplication is to be used (e.g., not Strassen)?*

There are an exponential number of different possible parenthesizations, in fact $\binom{2(n-1)}{n-1}/n$, so we don't want to search through all of them. Dynamic Programming gives us a better way.

As before, let's first think: how might we do this recursively? One way is that for each possible split for the final multiplication, recursively solve for the optimal parenthesization of the left and right sides, and calculate the total cost (the sum of the costs returned by the two recursive calls plus the ℓmn cost of the final multiplication, where "m" depends on the location of that split). Then take the overall best top-level split.

For Dynamic Programming, the key question is now: in the above procedure, as you go through the recursion, what do the subproblems look like and how many are there? Answer: each subproblem looks like "what is the best way to multiply some sub-interval of the matrices $A_i \times \ldots \times A_j$?" So, there are only $O(n^2)$ *different* subproblems.

The second question is now: how long does it take to solve a given subproblem assuming you've already solved all the smaller subproblems (i.e., how much time is spent inside any *given* recursive call)? Answer: to figure out how to best multiply $A_i \times \ldots \times A_j$, we just consider all possible middle points k and select the one that minimizes:

$$\begin{array}{rll} & \text{optimal cost to multiply } A_i \ldots A_k & \leftarrow \text{already computed} \\ + & \text{optimal cost to multiply } A_{k+1} \ldots A_j & \leftarrow \text{already computed} \\ + & \text{cost to multiply the results.} & \leftarrow \text{get this from the dimensions} \end{array}$$

This just takes $O(1)$ work for any given k, and there are at most n different values k to consider, so overall we just spend $O(n)$ time per subproblem. So, if we use Dynamic Programming to save our results in a lookup table, then since there are only $O(n^2)$ subproblems we will spend only $O(n^3)$ time overall.

If you want to do this using bottom-up Dynamic Programming, you would first solve for all subproblems with $j - i = 1$, then solve for all with $j - i = 2$, and so on, storing your results in an n by n matrix. The main difference between this problem and the two previous ones we have seen is that any *given* subproblem takes time $O(n)$ to solve rather than $O(1)$, which is why we get $O(n^3)$

total running time. It turns out that by being very clever you can actually reduce this to $O(1)$ amortized time per subproblem, producing an $O(n^2)$-time algorithm, but we won't get into that here.[3]

11.7 High-level discussion of Dynamic Programming

What kinds of problems can be solved using Dynamic Programming? One property these problems have is that if the optimal solution involves solving a subproblem, then it uses the *optimal* solution to that subproblem. For instance, say we want to find the shortest path from A to B in a graph, and say this shortest path goes through C. Then it must be using the shortest path from C to B. Or, in the knapsack example, if the optimal solution does not use item n, then it is the optimal solution for the problem in which item n does not exist. The other key property is that there should be only a polynomial number of different subproblems. These two properties together allow us to build the optimal solution to the final problem from optimal solutions to subproblems.

In the top-down view of dynamic programming, the first property above corresponds to being able to write down a recursive procedure for the problem we want to solve. The second property corresponds to making sure that this recursive procedure makes only a polynomial number of *different* recursive calls. In particular, one can often notice this second property by examining the arguments to the recursive procedure: e.g., if there are only two integer arguments that range between 1 and n, then there can be at most n^2 different recursive calls.

Sometimes you need to do a little work on the problem to get the optimal-subproblem-solution property. For instance, suppose we are trying to find paths between locations in a city, and some intersections have no-left-turn rules (this is particulatly bad in San Francisco). Then, just because the fastest way from A to B goes through intersection C, it doesn't necessarily use the fastest way to C because you might need to be coming into C in the correct direction. In fact, the right way to model that problem as a graph is not to have one node per intersection, but rather to have one node per $\langle$*intersection, direction*$\rangle$ pair. That way you recover the property you need.

[3]For details, see Knuth (insert ref).

Lecture 12

Graph Algorithms I

12.1 Overview

This is the first of several lectures on graph algorithms. We will see how simple algorithms like depth-first-search can be used in clever ways (for a problem known as *topological sorting*) and will see how Dynamic Programming can be used to solve problems of finding shortest paths. Topics in this lecture include:

- Basic notation and terminology for graphs.
- Depth-first-search for Topological Sorting.
- Dynamic-Programming algorithms for shortest path problems: Bellman-Ford (for single-source) and Floyd-Warshall (for all-pairs).

12.2 Introduction

Many algorithmic problems can be modeled as problems on graphs. Today we will talk about a few important ones and we will continue talking about graph algorithms for much of the rest of the course.

As a reminder of basic terminology: a graph is a set of *nodes* or *vertices*, with edges between some of the nodes. We will use V to denote the set of vertices and E to denote the set of edges. If there is an edge between two vertices, we call them *neighbors*. The *degree* of a vertex is the number of neighbors it has. Unless otherwise specified, we will not allow self-loops or multi-edges (multiple edges between the same pair of nodes). As is standard with discussing graphs, we will use $n = |V|$, and $m = |E|$, and we will let $V = \{1, \ldots, n\}$.

The above describes an *undirected* graph. In a *directed* graph, each edge now has a direction. For each node, we can now talk about out-neighbors (and out-degree) and in-neighbors (and in-degree). In a directed graph you may have both an edge from i to j and an edge from j to i.

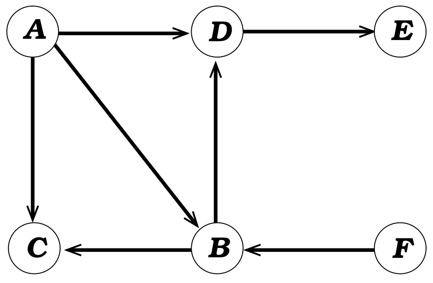
To make sure we are all on the same page, what is the maximum number of total edges in an *undirected* graph? Answer: $\binom{n}{2}$. What about a *directed* graph? Answer: $n(n-1)$.

There are two standard representations for graphs. The first is an *adjacency list*, which is an array of size n where $A[i]$ is the list of out-neighbors of node i. The second is an *adjacency matrix*, which is an n by n matrix where $A[i, j] = 1$ iff there is an edge from i to j. For an undirected graph, the adjacency matrix will be symmetric. Note that if the graph is reasonably sparse, then an adjacency list will be more compact than an adjacency matrix, because we are only implicitly representing the non-edges. In addition, an adjacency list allows us to access all edges out of some node v in time proportional to the out-degree of v. In general, however, the most convenient representation for a graph will depend on what we want to do with it.

We will also talk about weighted graphs where edges may have weights or costs on them. The best notion of an adjacency matrix for such graphs (e.g., should non-edges have weight 0 or weight infinity) will again depend on what problem we are trying to model.

12.3 Topological sorting and Depth-first Search

A **Directed Acyclic Graph (DAG)** is a directed graph without any cycles.[1] E.g.,



Given a DAG, the **topological sorting** problem is to find an ordering of the vertices such that all edges go forward in the ordering. A typical situation where this problem comes up is when you are given a set of tasks to do with precedence constraints (you need to do A and F before you can do B, etc.), and you want to find a legal ordering for performing the jobs. We will assume here that the graph is represented using an adjacency list.

One way to solve the topological sorting problem is to put all the nodes into a priority queue according to in-degree. You then repeatedly pull out the node of minimum in-degree (which should be zero — otherwise you output "graph is not acyclic") and then decrement the keys of each of its out-neighbors. Using a heap to implement the priority queue, this takes time $O(m \log n)$. However, it turns out there is a better algorithm: a simple but clever $O(m + n)$-time approach based on depth-first search.[2]

To be specific, by a *Depth-First Search (DFS) of a graph* we mean the following procedure. First, pick a node and perform a standard depth-first search from there. When that DFS returns, if the whole graph has not yet been visited, pick the next unvisited node and repeat the process.

[1]It would perhaps be more proper to call this an *acyclic directed graph*, but "DAG" is easier to say.

[2]You can also directly improve the first approach to $O(m + n)$ time by using the fact that the minimum always occurs at zero (think about how you might use that fact to speed up the algorithm). But we will instead examine the DFS-based algorithm because it is particularly elegant.

Continue until all vertices have been visited. Specifically, as pseudocode, we have:

```
DFSmain(G):
 For v=1 to n: if v is not yet visited, do DFS(v).

DFS(v):
  mark v as visited. // entering node v
  for each unmarked out-neighbor w of v: do DFS(w).
  return.            // exiting node v.
```

DFS takes time $O(1 + \text{out-degree}(v))$ per vertex v, for a total time of $O(m + n)$. Here is now how we can use this to perform a topological sorting:

1. Do depth-first search of G, outputting the nodes as you *exit* them.
2. Reverse the order of the list output in Step 1.

Claim 12.1 *If there is an edge from u to v, then v is exited first. (This implies that when we reverse the order, all edges point forward and we have a topological sorting.)*

Proof: [In this proof, think of $u = B$, and $v = D$ in the previous picture.] The claim is easy to see if our DFS entered node u before ever entering node v, because it will eventually enter v and then exit v before popping out of the recursion for DFS(u). But, what if we entered v first? In this case, we would exit v before even entering u since there cannot be a path from v to u (else the graph wouldn't be acyclic). So, that's it. ■

12.4 Shortest Paths

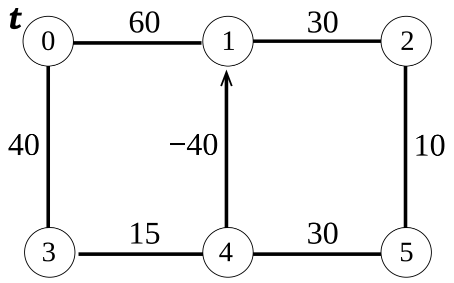
We are now going to turn to another basic graph problem: finding shortest paths in a weighted graph, and we will look at several algorithms based on Dynamic Programming. For an edge (i, j) in our graph, let's use $len(i, j)$ to denote its length. The basic shortest-path problem is as follows:

Definition 12.1 *Given a weighted, directed graph G, a start node s and a destination node t, the* **s-t shortest path** *problem is to output the shortest path from s to t. The* **single-source** *shortest path problem is to find shortest paths from s to every node in G. The (algorithmically equivalent)* **single-sink** *shortest path problem is to find shortest paths from every node in G to t.*

We will allow for negative-weight edges (we'll later see some problems where this comes up when using shortest-path algorithms as a subroutine) but will assume no negative-weight cycles (else the shortest path can wrap around such a cycle infinitely often and has length negative infinity). As a shorthand, if there is an edge of length ℓ from i to j and also an edge of length ℓ from j to i, we will often just draw them together as a single undirected edge. So, all such edges must have positive weight.

12.4.1 The Bellman-Ford Algorithm

We will now look at a Dynamic Programming algorithm called the Bellman-Ford Algorithm for the single-sink (or single-source) shortest path problem.[3] Let us develop the algorithm using the following example:



How can we use Dyanamic Programming to find the shortest path from all nodes to t? First of all, as usual for Dynamic Programming, let's just compute the *lengths* of the shortest paths first, and afterwards we can easily reconstruct the paths themselves. The idea for the algorithm is as follows:

1. For each node v, find the length of the shortest path to t that uses at most 1 edge, or write down ∞ if there is no such path.

   This is easy: if $v = t$ we get 0; if $(v, t) \in E$ then we get $len(v, t)$; else just put down ∞.

2. Now, suppose for all v we have solved for length of the shortest path to t that uses $i - 1$ or fewer edges. How can we use this to solve for the shortest path that uses i or fewer edges?

   Answer: the shortest path from v to t that uses i or fewer edges will first go to some neighbor x of v, and then take the shortest path from x to t that uses $i-1$ or fewer edges, which we've already solved for! So, we just need to take the min over all neighbors x of v.

3. How far do we need to go? Answer: at most $i = n - 1$ edges.

Specifically, here is pseudocode for the algorithm. We will use `d[v][i]` to denote the length of the shortest path from v to t that uses i or fewer edges (if it exists) and infinity otherwise ("d" for "distance"). Also, for convenience we will use a base case of $i = 0$ rather than $i = 1$.

Bellman-Ford pseudocode:

```
initialize d[v][0] = infinity for v != t.  d[t][i]=0 for all i.
for i=1 to n-1:
   for each v != t:
     d[v][i] = $\min_{(v,x)\in E}$ (len(v,x) + d[x][i-1])
For each v, output d[v][n-1].
```

Try it on the above graph!

We already argued for correctness of the algorithm. What about running time? The min operation takes time proportional to the out-degree of v. So, the inner for-loop takes time proportional to the sum of the out-degrees of all the nodes, which is $O(m)$. Therefore, the total time is $O(mn)$.

[3]Bellman is credited for inventing Dynamic Programming, and even if the technique can be said to exist inside some algorithms before him, he was the first to distill it as an important technique.

So far we have only calculated the *lengths* of the shortest paths; how can we reconstruct the paths themselves? One easy way is (as usual for DP) to work backwards: if you're at vertex v at distance $d[v]$ from t, move to the neighbor x such that $d[v] = d[x] + len(v, x)$. This allows us to reconstruct the path in time $O(m + n)$ which is just a low-order term in the overall running time.

12.5 All-pairs Shortest Paths

Say we want to compute the length of the shortest path between *every* pair of vertices. This is called the **all-pairs** shortest path problem. If we use Bellman-Ford for all n possible destinations t, this would take time $O(mn^2)$. We will now see two alternative Dynamic-Programming algorithms for this problem: the first uses the matrix representation of graphs and runs in time $O(n^3 \log n)$; the second, called the *Floyd-Warshall* algorithm uses a different way of breaking into subproblems and runs in time $O(n^3)$.

12.5.1 All-pairs Shortest Paths via Matrix Products

Given a weighted graph G, define the matrix $A = A(G)$ as follows:

- $A[i, i] = 0$ for all i.
- If there is an edge from i to j, then $A[i, j] = len(i, j)$.
- Otherwise, $A[i, j] = \infty$.

I.e., $A[i, j]$ is the length of the shortest path from i to j using 1 or fewer edges. Now, following the basic Dynamic Programming idea, can we use this to produce a new matrix B where $B[i, j]$ is the length of the shortest path from i to j using 2 or fewer edges?

Answer: yes. $B[i, j] = \min_k(A[i, k] + A[k, j])$. Think about why this is true!

I.e., what we want to do is compute a matrix product $B = A \times A$ except we change "*" to "+" and we change "+" to "min" in the definition. In other words, instead of computing the sum of products, we compute the min of sums.

What if we now want to get the shortest paths that use 4 or fewer edges? To do this, we just need to compute $C = B \times B$ (using our new definition of matrix product). I.e., to get from i to j using 4 or fewer edges, we need to go from i to some intermediate node k using 2 or fewer edges, and then from k to j using 2 or fewer edges.

So, to solve for all-pairs shortest paths we just need to keep squaring $O(\log n)$ times. Each matrix multiplication takes time $O(n^3)$ so the overall running time is $O(n^3 \log n)$.

12.5.2 All-pairs shortest paths via Floyd-Warshall

Here is an algorithm that shaves off the $O(\log n)$ and runs in time $O(n^3)$. The idea is that instead of increasing the number of edges in the path, we'll increase the set of vertices we allow as intermediate nodes in the path. In other words, starting from the same base case (the shortest path that uses no intermediate nodes), we'll then go on to considering the shortest path that's allowed to use node 1 as an intermediate node, the shortest path that's allowed to use $\{1, 2\}$ as intermediate nodes, and so on.

```
// After each iteration of the outside loop, A[i][j] = length of the
// shortest i->j path that's allowed to use vertices in the set 1..k
for k = 1 to n do:
  for each i,j do:
    A[i][j] = min( A[i][j], (A[i][k] + A[k][j]);
```

I.e., you either go through node k or you don't. The total time for this algorithm is $O(n^3)$. What's amazing here is how compact and simple the code is!

Lecture 13

Graph Algorithms II

13.1 Overview

In this lecture we begin with one more algorithm for the shortest path problem, *Dijkstra's algorithm.* We then will see how the basic approach of this algorithm can be used to solve other problems including finding *maximum bottleneck paths* and the *minimum spanning tree* (MST) problem. We will then expand on the minimum spanning tree problem, giving one more algorithm, *Kruskal's algorithm*, which to implement efficiently requires an good data structure for something called the *union-find problem.* Topics in this lecture include:

- Dijkstra's algorithm for shortest paths when no edges have negative weight.
- The Maximum Bottleneck Path problem.
- Minimum Spanning Trees: Prim's algorithm and Kruskal's algorithm.

13.2 Shortest paths revisited: Dijkstra's algorithm

Recall the *single-source* shortest path problem: given a graph G, and a start node s, we want to find the shortest path from s to all other nodes in G. These shortest paths can all be described by a tree called the *shortest path tree* from start node s.

Definition 13.1 *A* **Shortest Path Tree** *in G from start node s is a tree (directed outward from s if G is a directed graph) such that the shortest path in G from s to any destination vertex t is the path from s to t in the tree.*

Why must such a tree exist? The reason is that if the shortest path from s to t goes through some intermediate vertex v, then it must use a shortest path from s to v. Thus, every vertex $t \neq s$ can be assigned a "parent", namely the second-to-last vertex in this path (if there are multiple equally-short paths, pick one arbitrarily), creating a tree. In fact, the Bellman-Ford Dynamic-Programming algorithm from the last class was based on this "optimal subproblem" property.

The first algorithm for today, *Dijkstra's algorithm*, builds the tree outward from s in a greedy fashion. Dijkstra's algorithm is faster than Bellman-Ford. However, it requires that all edge lengths be non-negative. *See if you can figure out where the proof of correctness of this algorithm requires non-negativity.*

We will describe the algorithm the way one views it conceptually, rather than the way one would code it up (we will discuss that after proving correctness).

Dijkstra's Algorithm:

Input: Graph G, with each edge e having a length $len(e)$, and a start node s.

Initialize: tree = $\{s\}$, no edges. Label s as having distance 0 to itself.

Invariant: nodes in the tree are labeled with the correct distance to s.

Repeat:

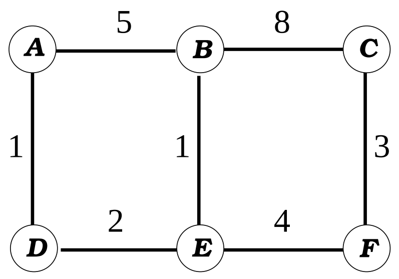
1. For each neighbor x of the tree, compute an (over)-estimate of its distance to s:

$$\text{distance}(x) = \min_{e=(v,x):v\in\text{tree}} [\text{distance}(v) + len(e)] \tag{13.1}$$

   In other words, by our invariant, this is the length of the shortest path to x whose only edge not in the tree is the very last edge.

2. Insert the node x of minimum distance into tree, connecting it via the argmin (the edge e used to get distance(x) in the expression (13.1)).

Let us run the algorithm on the following example starting from vertex A:



Theorem 13.1 *Dijkstra's algorithm correctly produces a shortest path tree from start node s. Specifically, even if some of distances in step 1 are too large, the* minimum *one is correct.*

Proof: Say x is the neighbor of the tree of smallest distance(x). Let P denote the *true* shortest path from s to x, choosing the one with the fewest non-tree edges if there are ties. What we need to argue is that the last edge in P must come directly from the tree. Let's argue this by contradiction. Suppose instead the first non-tree vertex in P is some node $y \neq x$. Then, the length of P must be at least distance(y), and by definition, distance(x) is smaller (or at least as small if there is a tie). This contradicts the definition of P. ■

Did you catch where "non-negativity" came in in the proof? Can you find an example with negative-weight directed edges where Dijkstra's algorithm actually fails?

Running time: To implement this efficiently, rather than recomputing the distances every time in step 1, you simply want to update the ones that actually are affected when a new node is added to the tree in step 2, namely the neighbors of the node added. If you use a heap data structure to store the neighbors of the tree, you can get a running time of $O(m \log n)$. In particular, you can start by giving all nodes a distance of infinity except for the start with a distance of 0, and putting all nodes into a min-heap. Then, repeatedly pull off the minimum and update its neighbors, tentatively assigning parents whenever the distance of some node is lowered. It takes linear time to initialize the heap, and then we perform m updates at a cost of $O(\log n)$ each for a total time of $O(m \log n)$.

If you use something called a "Fibonacci heap" (that we're not going to talk about) you can actually get the running time down to $O(m+n \log n)$. The key point about the Fibonacci heap is that while it takes $O(\log n)$ time to remove the minimum element just like a standard heap (an operation we perform n times), it takes only amortized $O(1)$ time to decrease the value of any given key (an operation we perform m times).

13.3 Maximum-bottleneck path

Here is another problem you can solve with this type of algorithm, called the "maximum bottleneck path" problem. Imagine the edge weights represent capacities of the edges ("widths" rather than "lengths") and you want the path between two nodes whose minimum width is largest. How could you modify Dijkstra's algorithm to solve this?

To be clear, define the *width* of a path to be the minimum width of any edge on the path, and for a vertex v, define widthto(v) to be the width of the widest path from s to v (say that widthto(s) $= \infty$). To modify Dijkstra's algorithm, we just need to change the update rule to:

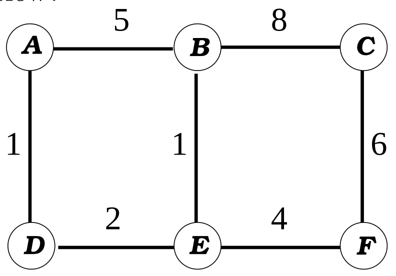
$$\text{widthto}(x) = \max_{e=(v,x):v\in\text{tree}} [\min(\text{widthto}(v), \text{width(e)})]$$

and now put the node x of *maximum* "widthto" into tree, connecting it via the argmax. We'll actually use this later in the course.

13.4 Minimum Spanning Trees

A **spanning tree** of a graph is a tree that touches all the vertices (so, it only makes sense in a connected graph). A **minimum spanning tree** (MST) is a spanning tree whose sum of edge lengths is as short as possible (there may be more than one). We will sometimes call the sum of edge lengths in a tree the *size* of the tree. For instance, imagine you are setting up a communication network among a set of sites and you want to use the least amount of wire possible. *Note:* our definition is only for *undirected* graphs.

What is the MST in the graph below?



13.4.1 Prim's algorithm

Prim's algorithm is an MST algorithm that works much like Dijkstra's algorithm does for shortest path trees. In fact, it's even simpler (though the correctness proof is a bit trickier).

Prim's Algorithm:

1. Pick some arbitrary start node s. Initialize tree $T = \{s\}$.
2. Repeatedly add the shortest edge incident to T (the shortest edge having one vertex in T and one vertex not in T) until the tree spans all the nodes.

So the algorithm is the same as Dijsktra's algorithm, except you don't add distance(v) to the length of the edge when deciding which edge to put in next. For instance, what does Prim's algorithm do on the above graph?

Before proving correctness for the algorithm, we first need a useful fact about spanning trees: if you take any spanning tree and add a new edge to it, this creates a cycle. The reason is that there already was one path between the endpoints (since it's a *spanning* tree), and now there are two. If you then remove any edge in the cycle, you get back a spanning tree (removing one edge from a cycle cannot disconnect a graph).

Theorem 13.2 *Prim's algorithm correctly finds a minimum spanning tree of the given graph.*

Proof: We will prove correctness by induction. Let G be the given graph. Our inductive hypothesis will be that the tree T constructed so far is consistent with (is a subtree of) some minimum spanning tree M of G. This is certainly true at the start. Now, let e be the edge chosen by the algorithm. We need to argue that the new tree, $T \cup \{e\}$ is also consistent with some minimum spanning tree M' of G. If $e \in M$ then we are done ($M' = M$). Else, we argue as follows.

Consider adding e to M. As noted above, this creates a cycle. Since e has one endpoint in T and one outside T, if we trace around this cycle we must eventually get to an edge e' that goes back in to T. We know $len(e') \geq len(e)$ by definition of the algorithm. So, if we add e to M and remove e', we get a new tree M' that is no larger than M was and contains $T \cup \{e\}$, maintaining our induction and proving the theorem. ■

Running time: We can implement this in the same was as Dijkstra's algorithm, getting an $O(m \log n)$ running time if we use a standard heap, or $O(m + n \log n)$ running time if we use a Fibonacci heap. The only difference with Dijkstra's algorithm is that when we store the neighbors of T in a heap, we use priority values equal to the shortest edge connecting them to T (rather than the smallest sum of "edge length plus distance of endpoint to s").

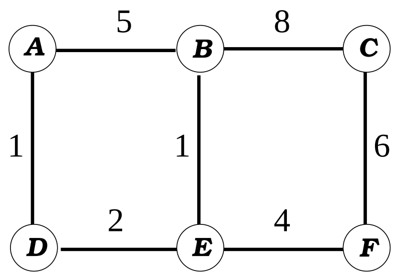
13.4.2 Kruskal's algorithm

Here is another algorithm for finding minimum spanning trees called Kruskal's algorithm. It is also greedy but works in a different way.

Kruskal's Algorithm:

Sort edges by length and examine them from shortest to longest. Put each edge into the current forest (a forest is just a set of trees) if it doesn't form a cycle with the edges chosen so far.

E.g., let's look at how it behaves in the graph below:



Kruskal's algorithm sorts the edges and then puts them in one at a time so long as they don't form a cycle. So, first the AD and BE edges will be added, then the DE edge, and then the EF edge. The AB edge will be skipped over because it forms a cycle, and finally the CF edge will be added (at that point you can either notice that you have included $n-1$ edges and therefore are done, or else keep going, skipping over all the remaining edges one at a time).

Theorem 13.3 *Kruskal's algorithm correctly finds a minimum spanning tree of the given graph.*

Proof: We can use a similar argument to the one we used for Prim's algorithm. Let G be the given graph, and let F be the forest we have constructed so far (initially, F consists of n trees of 1 node each, and at each step two trees get merged until finally F is just a single tree at the end). Assume by induction that there exists an MST M of G that is consistent with F, i.e., all edges in F are also in M; this is clearly true at the start when F has no edges. Let e be the next edge added by the algorithm. Our goal is to show that there exists an MST M' of G consistent with $F \cup \{e\}$.

If $e \in M$ then we are done ($M' = M$). Else add e into M, creating a cycle. Since the two endpoints of e were in different trees of F, if you follow around the cycle you must eventually traverse some edge $e' \neq e$ whose endpoints are also in two different trees of F (because you eventually have to get back to the node you started from). Now, both e and e' were eligible to be added into F, which by definition of our algorithm means that $len(e) \leq len(e')$. So, adding e and removing e' from M creates a tree M' that is also a MST and contains $F \cup \{e\}$, as desired. ■

Running time: The first step is sorting the edges by length which takes time $O(m \log m)$. Then, for each edge we need to test if it connects two different components. This seems like it should be a real pain: how can we tell if an edge has both endpoints in the same component? It turns out there's a nice data structure called the *Union-Find*' data structure for doing this operation. It is so efficient that it actually will be a low-order cost compared to the sorting step.

We will talk about the union-find problem in the next class, but just as a preview, the *simpler* version of that data structure takes time $O(m+n \log n)$ for our series of operations. This is already good enough for us, since it is low-order compared to the sorting time. There is also a more sophisticated version, however, whose total time is $O(m \lg^* n)$, in fact $O(m\alpha(n))$, where $\alpha(n)$ is the inverse-Ackermann function that grows even more slowly than $\lg^*$.

What is $\lg^*$? $\lg^*(n)$ is the number of times you need to take $\log_2$ until you get down to 1. So,

$$\begin{aligned} \lg^*(2) &= 1 \\ \lg^*(2^2 = 4) &= 2 \\ \lg^*(2^4 = 16) &= 3 \\ \lg^*(2^{16} = 65536) &= 4 \\ \lg^*(2^{65536}) &= 5. \end{aligned}$$

I won't define Ackerman, but to get $\alpha(n)$ up to 5, you need n to be at least a stack of 256 2's.

Lecture 14

Midterm

Midterm today. Good luck everyone!

Lecture 15

Graph Algorithms III: Union-Find

15.1 Overview

In this lecture we describe the *union-find* problem. This is a problem that captures the key task we had to solve in order to efficiently implement Kruskal's algorithm. We then give two data structures for it with good amortized running time.

15.2 Motivation

To motivate the union-find problem, let's recall Kruskal's minimum spanning tree algorithm.

Kruskal's Algorithm (recap):
Sort edges by length and examine them from shortest to longest. Put each edge into the current forest if it doesn't form a cycle with the edges chosen so far.

We argued correctness last time. Today, our concern is running time. The initial step takes time $O(|E| \log |E|)$ to sort. Then, for each edge, we need to test if it connects two different components. If it does, we will insert the edge, merging the two components into one; if it doesn't (the two endpoints are in the same component), then we will skip this edge and go on to the next edge. So, to do this efficiently we need a data structure that can support the basic operations of (a) determining if two nodes are in the same component, and (b) merging two components together. This is the union-find problem.

15.3 The Union-Find Problem

The general setting for the union-find problem is that we are maintaining a collection of disjoint sets $\{S_1, S_2, \ldots, S_k\}$ over some universe, with the following operations:

MakeSet(x)**:** create the set $\{x\}$.

Union(x, y)**:** replace the set x is in (let's call it S) and the set y is in (let's call it S') with the single set $S \cup S'$.

Find(x): return the unique ID for the set containing x (this is just some representative element of this set).

Given these operations, we can implement Kruskal's algorithm as follows. The sets S_i will be the sets of vertices in the different trees in our forest. We begin with MakeSet(v) for all vertices v (every vertex is in its own tree). When we consider some edge (v, w) in the algorithm, we just test whether Find(v) equals Find(w). If they are equal, it means that v and w are already in the same tree so we skip over the edge. If they are not equal, we insert the edge into our forest and perform a Union(v, w) operation. All together we will do $|V|$ MakeSet operations, $|V| - 1$ Unions, and $2|E|$ Find operations.

Notation and Preliminaries: in the discussion below, it will be convenient to define n as the number of MakeSet operations and m as the total number of operations (this matches the number of vertices and edges in the graph up to constant factors, and so is a reasonable use of n and m). Also, it is easiest to think conceptually of these data structures as adding fields to the items themselves, so there is never an issue of "how do I locate a given element v in the structure?".

15.4 Data Structure 1 (list-based)

Our first data structure is a simple one with a very cute analysis. The total cost for the operations will be $O(m + n \log n)$.

In this data structure, the sets will be just represented as linked lists: each element has a pointer to the next element in its list. However, we will augment the list so that each element also has a pointer directly to head of its list. The head of the list is the representative element. We can now implement the operations as follows:

MakeSet(x): just set `x->head=x`. This takes constant time.

Find(x): just return `x->head`. Also takes constant time.

Union(x, y): To perform a union operation we merge the two lists together, and reset the head pointers on one of the lists to point to the head of the other.

Let A be the list containing x and B be the list containing y, with lengths L_A and L_B respectively. Then we can do this in time $O(L_A + L_B)$ by appending B onto the end of A as follows. We first walk down A to the end, and set the final `next` pointer to point to `y->head`. This takes time $O(L_A)$. Next we go to `y->head` and walk down B, resetting head pointers of elements in B to point to `x->head`. This takes time $O(L_B)$.

Can we reduce this to just $O(L_B)$? Yes. Instead of appending B onto the end of A, we can just splice B into the middle of A, at x. I.e., let `z=x->next`, set `x->next=y->head`, then walk down B as above, and finally set the final `next` pointer of B to `z`.

Can we reduce this to $O(\min(L_A, L_B))$? Yes. Just store the length of each list in the head. Then compare and insert the shorter list into the middle of the longer one. Then update the length count to $L_A + L_B$.

We now prove this simple data structure has the running time we wanted.

Theorem 15.1 *The above algorithm has total running time $O(m + n \log n)$.*

Proof: The Find and MakeSet operations are constant time so they are covered by the $O(m)$ term. Each Union operation has cost proportional to the length of the list whose head pointers get updated. So, we need to find some way of analyzing the total cost of the Union operations.

Here is the key idea: we can pay for the union operation by charging $O(1)$ to each element whose head pointer is updated. So, all we need to do is sum up the costs charged to all the elements over the entire course of the algorithm. Let's do this by looking from the point of view of some lowly element x. Over time, how many times does x get walked on and have its head pointer updated? The answer is that its head pointer is updated at most $\log n$ times. The reason is that we only update head pointers on the *smaller* of the two lists being joined, so every time x gets updated, the size of the list it is in at least doubles, and this can happen at most $\log n$ times. So, we were able to pay for unions by charging the elements whose head pointers are updated, and no element gets charged more than $O(\log n)$ total, so the total cost for unions is $O(n \log n)$, or $O(m + n \log n)$ for all the operations together. ■

Recall that this is already low-order compared to the $O(m \log m)$ sorting time for Kruskal's algorithm.

15.5 Data Structure 2 (tree-based)

Even though the running time of the list-based data structure was pretty fast, let's think of ways we could make it even faster. One idea is that instead of updating all the head pointers in list B (or whichever was shorter) when we perform a Union, we could do this in a lazy way, just pointing the head of B to the head of A and then waiting until we actually perform a find operation on some item x before updating its pointer. This will decrease the cost of the Union operations but will increase the cost of Find operations because we may have to take multiple hops. Notice that by doing this we no longer need the downward pointers: what we have in general is a collection of trees, with all links pointing *up*. Another idea is that rather than deciding which of the two heads (or roots) should be the new one based on the *size* of their sets, perhaps there is some other quantity that would give us better performance. In particular, it turns out we can do better by setting the new root based on which tree has larger *rank*, which we will define in a minute.

We will prove that by implementing the two optimizations described above (lazy updates and union-by-rank), the total cost is bounded above by $O(m \lg^* n)$, where recall that $\lg^* n$ is the number of times you need to take $\log_2$ until you get down to 1. For instance,

$$\lg^*(2^{65536}) = 1 + \lg^*(65536) = 2 + \lg^*(16) = 3 + \lg^*(4) = 4 + \lg^*(2) = 5.$$

So, basically, $\lg^* n$ is never bigger than 5. Technically, the running time of this algorithm is even better: $O(m\alpha(m, n))$ where α is the inverse-Ackermann function which grows even more slowly than $\lg^*$. But the $\lg^* n$ bound is hard enough to prove — let's not go completely overboard!

We now describe the procedure more specifically. Each element (node) will have two fields: a *parent* pointer that points to its parent in its tree (or itself if it is the root) and a rank, which is an integer used to determine which node becomes the new root in a Union operation. The operations are as follows.

MakeSet(x)**:** set x's rank to 0 and its parent pointer to itself. This takes constant time.

Find(x)**:** starting from x, follow the parent pointers until you reach the root, updating x and all the nodes we pass over to point to the root. This is called *path compression.*

The running time for Find(x) is proportional to (original) distance of x to its root.

Union(x, y)**:** Let Union(x, y) = Link(Find(x), Find(y)), where Link(root1,root2) behaves as follows. If the one of the roots has larger rank than the other, then that one becomes the new root, and the other (smaller rank) root has its parent pointer updated to point to it. If the two roots have *equal* rank, then one of them (arbitrarily) is picked to be the new root *and its rank is increased by 1.* This procedure is called *union by rank.*

Properties of ranks: To help us understand this procedure, let's first develop some properties of ranks.

1. The rank of a node is the same as what the height of its subtree would be if we didn't do path compression. This is easy to see: if you take two trees of *different* heights and join them by making the root of the shorter tree into a child of the root of the taller tree, the heights do not change, but if the trees were the *same* height, then the final tree will have its height increase by 1.
2. If x is not a root, then rank(x) is strictly less than the rank of x's parent. We can see this by induction: the Union operation maintains this property, and the Find operation only increases the difference between the ranks of nodes and their parents.
3. The rank of a node x can only change if x is a root. Furthermore, once a node becomes a non-root, it is never a root again. These are immediate from the algorithm.
4. There are at most $n/2^r$ nodes of rank $\geq r$. The reason is that when a (root) node first reaches rank r, its tree must have at least 2^r nodes (this is easy to see by induction). Furthermore, by property 2, all the nodes in its tree (except for itself) have rank $< r$, and their ranks are never going to change by property 3. This means that (a) for each node x of rank $\geq r$, we can identify a set S_x of at least $2^r - 1$ nodes of smaller rank, and (b) for any two nodes x and y of rank $\geq r$, the sets S_x and S_y are disjoint. Since there are n nodes total, this implies there can be at most $n/2^r$ nodes of rank $\geq r$.

We're now ready to prove the following theorem.

Theorem 15.2 *The above tree-based algorithm has total running time $O(m \lg^* n)$.*

Proof: Let's begin with the easy parts. First of all, the Union does two Find operations plus a constant amount of extra work. So, we only need to worry about the time for the (at most $2m$) Find operations. Second, we can count the cost of a Find operation by charging \$1 for each parent pointer examined. So, when we do a Find(x), if x was a root then pay \$1 (just a constant, so that's ok). If x was a child of a root we pay \$2 (also just a constant, so that's ok also). If x was lower, then the *very rough* idea is that (except for the last \$2) every dollar we spend is shrinking the tree because of our path compression, so we'll be able to amortize this cost somehow. For the remaining part of the proof, we're only going to worry about the steps taken in a Find(x) operation up until we reach the child of the root, since the remainder is just constant time per operation. We'll analyze this using the ranks, and the properties we figured out above.

Step 1: let's imagine putting non-root nodes into buckets according to their rank. Bucket 0 contains all non-root nodes of rank 0, bucket 1 has all of rank 1, bucket 2 has ranks 2 through $2^2 - 1$, bucket 3 has ranks 2^2 through $2^{2^2} - 1$, bucket 4 has ranks 2^{2^2} through $2^{2^{2^2}} - 1$, etc. In general, a bucket has ranks r through $2^r - 1$. In total, we have $O(\lg^* n)$ buckets.

The point of this definition is that the number of nodes with rank in any given bucket is at most $n/(\text{upper bound of bucket} + 1)$, by property (4) of ranks above.

Step 2: When we walk up the tree in the Find(x) operation, we need to charge our steps to something. Here's the rule we will use: if the step we take moves us from a node u to a node v such that v is in the *same* bucket as u, then we charge it to node u (the walk-ee). But if v is in a higher bucket, we charge that step to x (the walk-er).

The easy part of this is the charge to x. We can move up in buckets at most $O(\lg^* n)$ times, since there are only $O(\lg^* n)$ different buckets. So, the total cost charged to the walk-ers (adding up over the m Find operations) is at most $O(m \lg^* n)$.

The harder part is the charge to the walk-ee. The point here is that first of all, the node u charged is not a root, so its rank is never going to change. Secondly, every time we charge this node u, the rank of its new parent (after path compression) is at least 1 larger than the rank of its previous parent by property 2. Now, it is true that increasing the rank of u's parent could conceivably happen $\log n$ times, which to us is a "big" number. *But*, once its parent's rank becomes large enough that it is in the next bucket, we never charge node u again as walk-ee. So, the bottom line is that the maximum charge any node u gets is the range of his bucket.

To finish the proof, we just said that a bucket of upper bound $B - 1$ has at most n/B elements in it, and its range is $\leq B$. So, the total charge to all walk-ees in this bucket over the entire course of the algorithm is at most $(n/B)B = n$. (Remember that the only elements being charged are non-roots, so once they start getting charged, their rank is fixed so they can't jump to some other bucket and start getting charged there too.) This means the *total* charge of this kind summed *over all buckets* is at most n times the number of buckets which is $O(n \lg^* n)$. ■ (Whew!)

Lecture 16

Network Flow I

16.1 Overview

In these next two lectures we are going to talk about an important algorithmic problem called the *Network Flow Problem*. Network flow is important because it can be used to express a wide variety of different kinds of problems. So, by developing good algorithms for solving network flow, we immediately will get algorithms for solving many other problems as well. In Operations Research there are entire courses devoted to network flow and its variants. Topics in today's lecture include:

- The definition of the network flow problem
- The basic Ford-Fulkerson algorithm
- The maxflow-mincut theorem
- The bipartite matching problem

16.2 The Network Flow Problem

We begin with a definition of the problem. We are given a directed graph G, a start node s, and a sink node t. Each edge e in G has an associated non-negative *capacity* $c(e)$, where for all non-edges it is implicitly assumed that the capacity is 0. For example, consider the graph in Figure 16.1 below.

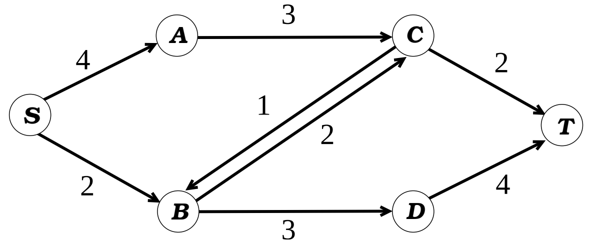


Figure 16.1: A network flow graph.

Our goal is to push as much *flow* as possible from s to t in the graph. The rules are that no edge can have flow exceeding its capacity, and for any vertex except for s and t, the flow *in* to the vertex must equal the flow *out* from the vertex. That is,

Capacity constraint: On any edge e we have $f(e) \leq c(e)$.

Flow conservation: For any vertex $v \notin \{s,t\}$, flow in equals flow out: $\sum_u f(u,v) = \sum_u f(v,u)$.

Subject to these constraints, we want to maximize the total flow into t. For instance, imagine we want to route message traffic from the source to the sink, and the capacities tell us how much bandwidth we're allowed on each edge.

E.g., in the above graph, what is the maximum flow from s to t? Answer: 5. Using "capacity[flow]" notation, the positive flow looks as in Figure 16.2. Note that the flow can split and rejoin itself.

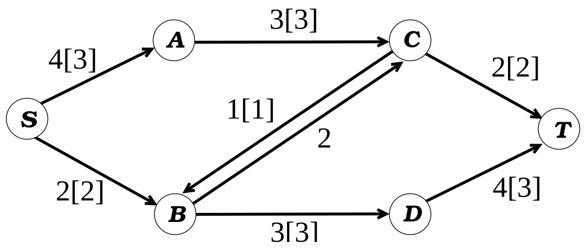


Figure 16.2: A network flow graph with positive flow shown using "capacity[flow]" notation.

How can you see that the above flow was really maximum? Notice, this flow saturates the $a \to c$ and $s \to b$ edges, and, if you remove these, you disconnect t from s. In other words, the graph has an "s-t cut" of size 5 (a set of edges of total capacity 5 such that if you remove them, this disconnects the source from the sink). The point is that any unit of flow going from s to t must take up at least 1 unit of capacity in these pipes. So, we know we're optimal.

We just argued that in general, the maximum s-t flow $\leq$ the capacity of the minimum s-t cut. An important property of flows, that we will prove as a byproduct of analyzing an algorithm for finding them, is that the maximum s-t flow is in fact *equal to* the capacity of the minimum s-t cut. This is called the *Maxflow-Mincut Theorem*. In fact, the algorithm will find a flow of some value k and a cut of capacity k, which will be proofs that both are optimal!

To describe the algorithm and analysis, it will help to be a bit more formal about a few of these quantities.

Definition 16.1 *An s-t* **cut** *is a set of edges whose removal disconnects t from s. Or, formally, a cut is a partition of the vertex set into two pieces A and B where $s \in A$ and $t \in B$. (The edges of the cut are then all edges going from A to B).*

Definition 16.2 *The* **capacity** *of a cut (A, B) is the sum of capacities of edges in the cut. Or, in the formal viewpoint, it is the sum of capacities of all edges going from A to B. (Don't include the edges from B to A.)*

Definition 16.3 *It will also be mathematically convenient for any edge (u,v) to define $f(v,u) = -f(u,v)$. This is called* **skew-symmetry**. *(We will think of flowing 1 unit on the edge from u to v as equivalently flowing -1 units on the back-edge from v to u.)*

The skew-symmetry convention makes it especially easy to add two flows together. For instance, if we have one flow with 1 unit on the edge (c, b) and another flow with 2 units on the edge (b, c), then adding them edge by edge does the right thing, resulting in a net flow of 1 unit from b to c. Also, using skew-symmetry, the total flow *out* of a node will always be the negative of the total flow *into* a node, so if we wanted we could rewrite the flow conservation condition as $\sum_u f(u, v) = 0$.

How can we find a maximum flow and prove it is correct? Here's a very natural strategy: find a path from s to t and push as much flow on it as possible. Then look at the leftover capacities (an important issue will be how exactly we define this, but we will get to it in a minute) and repeat. Continue until there is no longer any path with capacity left to push any additional flow on. Of course, we need to *prove* that this works: that we can't somehow end up at a suboptimal solution by making bad choices along the way. This approach, with the correct definition of "leftover capacity", is called the Ford-Fulkerson algorithm.

16.3 The Ford-Fulkerson algorithm

The Ford-Fulkerson algorithm is simply the following: while there exists an $s \to t$ path P of positive *residual capacity* (defined below), push the maximum possible flow along P. By the way, these paths P are called *augmenting paths*, because you use them to augment the existing flow.

Residual capacity is just the capacity left over given the existing flow, where we will use skew-symmetry to capture the notion that if we push f units of flow on an edge (u, v), this *increases* our ability to push flow on the back-edge (v, u) by f.

Definition 16.4 *Given a flow f in graph G, the* **residual capacity** $c_f(u, v)$ *is defined as* $c_f(u, v) = c(u, v) - f(u, v)$*, where recall that by skew-symmetry we have* $f(v, u) = -f(u, v)$.

For example, given the flow in Figure 16.2, the edge (s, a) has residual capacity 1. The back-edge (a, s) has residual capacity 3, because its original capacity was 0 and we have $f(a, s) = -3$.

Definition 16.5 *Given a flow f in graph G, the* **residual graph** G_f *is the directed graph with all edges of positive residual capacity, each one labeled by its residual capacity. Note: this may include back-edges of the original graph G.*

Let's do an example. Consider the graph in Figure 16.1 and suppose we push two units of flow on the path $s \to b \to c \to t$. We then end up with the following residual graph:

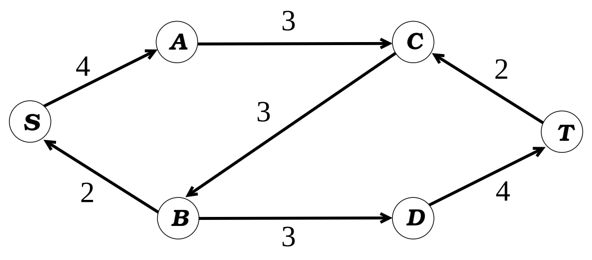


Figure 16.3: Residual graph resulting from pushing 2 units of flow along the path s-b-c-t in the graph in Figure 16.1.

If we continue running Ford-Fulkerson, we see that in this graph the only path we can use to augment the existing flow is the path $s \to a \to c \to b \to d \to t$. Pushing the maximum 3 units on this path we then get the next residual graph, shown in Figure 16.4. At this point there is no

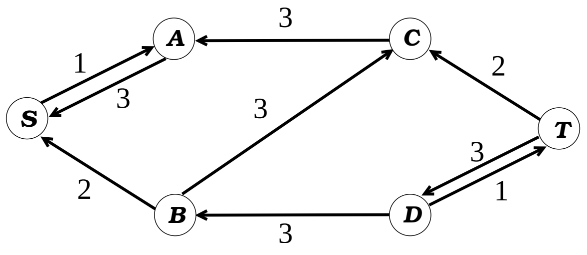


Figure 16.4: Residual graph resulting from pushing 3 units of flow along the path *s-a-c-b-d-t* in the graph in Figure 16.3.

longer a path from s to t so we are done.

We can think of Ford-Fulkerson as at each step finding a new flow (along the augmenting path) and adding it to the existing flow, where by *adding* two flows we mean adding them edge by edge. Notice that the sum of two flows continues to satisfy flow-conservation and skew-symmetry. The definition of residual capacity ensures that the flow found by Ford-Fulkerson is *legal* (doesn't exceed the capacity constraints in the original graph). We now need to prove that in fact it is *maximum.* We'll worry about the number of iterations it takes and how to improve that later.

Note that one nice property of the residual graph is that it means that at each step we are left with same type of problem we started with. So, to implement Ford-Fulkerson, we can use any black-box path-finding method (e.g., DFS).

Theorem 16.1 *The Ford-Fulkerson algorithm finds a maximum flow.*

Proof: Let's look at the final residual graph. This graph must have s and t disconnected by definition of the algorithm. Let A be the component containing s and B be the rest. Let c be the capacity of the (A, B) cut in the *original* graph — so we know we can't do better than c.

The claim is that we in fact *did* find a flow of value c (which therefore implies it is maximum). Here's why: let's look at what happens to the residual capacity of the (A, B) cut after each iteration of the algorithm. Say in some iteration we found a path with k units of flow. Then, even if the path zig-zagged between A and B, every time we went from A to B we added k to the flow from A to B and subtracted k from the residual capacity of the (A, B) cut, and every time we went from B to A we took away k from this flow and added k to the residual capacity of the cut[1]; moreover, we must have gone from A to B *exactly* one more time than we went from B to A. So, the residual capacity of this cut went down by exactly k. So, the drop in capacity is equal to the increase in flow. Since at the end the residual capacity is zero (remember how we defined A and B) this means the total flow is equal to c.

So, we've found a flow of value *equal* to the capacity of this cut. We know we can't do better, so this must be a max flow, and (A, B) must be a minimum cut. ■

[1]This is where we use the fact that if we flow k units on the edge (u, v), then in addition to reducing the residual capacity of the (u, v) edge by k we also *add* k to the residual capacity of the back-edge (v, u).

Notice that in the above argument we actually proved the nonobvious *maxflow-mincut* theorem:

Theorem 16.2 *In any graph G, for any two vertices s and t, the maximum flow from s to t equals the capacity of the minimum (s,t)-cut.*

We have also proven the *integral-flow theorem*: if all capacities are integers, then there is a maximum flow in which all flows are integers. This seems obvious, but you'll use it to show something that's not at all obvious on homework 5!

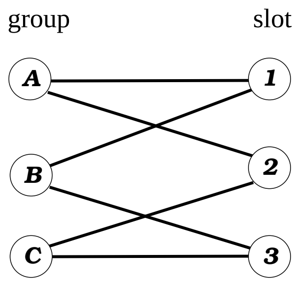
In terms of running time, if all capacities are integers and the maximum flow is F, then the algorithm makes at most F iterations (since each iteration pushes at least one more unit of flow from s to t). We can implement each iteration in time $O(m+n)$ using DFS. So we get the following result.

Theorem 16.3 *If the given graph G has integer capacities, Ford-Fulkerson runs in time $O(F(m+n))$ where F is the value of the maximum s-t flow.*

In the next lecture we will look at methods for reducing the number of iterations the algorithm can take. For now, let's see how we can use an algorithm for the max flow problem to solve other problems as well: that is, how we can *reduce* other problems to the one we now know how to solve.

16.4 Bipartite Matching

Say we wanted to be more sophisticated about assigning groups to homework presentation slots. We could ask each group to list the slots acceptable to them, and then write this as a bipartite graph by drawing an edge between a group and a slot if that slot is acceptable to that group. For example:



This is an example of a **bipartite graph**: a graph with two sides L and R such that all edges go between L and R. A **matching** is a set of edges with no endpoints in common. What we want here in assigning groups to time slots is a **perfect matching**: a matching that connects every point in L with a point in R. For example, what is a perfect matching in the bipartite graph above?

More generally (say there is no perfect matching) we want a **maximum matching**: a matching with the maximum possible number of edges. We can solve this as follows:

Bipartite Matching:

1. Set up a fake "start" node s connected to all vertices in L. Connect all vertices in R to a fake "sink" node T. Orient all edges left-to-right and give each a capacity of 1.

2. Find a max flow from s to t using Ford-Fulkerson.

3. Output the edges between L and R containing nonzero flow as the desired matching.

This finds a legal matching because edges from R to t have capacity 1, so the flow can't use two edges *into* the same node, and similarly the edges from s to L have capacity 1, so you can't have flow on two edges *leaving* the same node in L. It's a *maximum* matching because any matching gives you a flow of the same value: just connect s to the heads of those edges and connect the tails of those edges to t. (So if there was a better matching, we wouldn't be at a maximum flow).

What about the number of iterations of path-finding? This is at most the number of edges in the matching since each augmenting path gives us one new edge.

Let's run the algorithm on the above example. Notice a neat fact: say we start by matching A to 1 and C to 3. These are bad choices, but the augmenting path *automatically* undoes them as it improves the flow!

Matchings come up in many different problems like matching up suppliers to customers, or cell-phones to cell-stations when you have overlapping cells. They are also a basic part of other algorithmic problems.

Lecture 17

Network Flow II

17.1 Overview

The Ford-Fulkerson algorithm discussed in the last class takes time $O(F(n+m))$, where F is the value of the maximum flow, when all capacities are integral. This is fine if all edge capacities are small, but if they are large numbers written in binary then this could even be exponentially large in the description size of the problem. In this lecture we examine some improvements to the Ford-Fulkerson algorithm that produce much better (polynomial) running times. We then consider a generalization of max-flow called the min-cost max flow problem. Specific topics covered include:

- Edmonds-Karp Algorithm #1
- Edmonds-Karp Algorithm #2
- Further improvements
- Min-cost max flow

17.2 Network flow recap

Recall that in the network flow problem we are given a directed graph G, a source s, and a sink t. Each edge (u, v) has some capacity $c(u, v)$, and our goal is to find the maximum flow possible from s to t.

Last time we looked at the Ford-Fulkerson algorithm, which we used to prove the maxflow-mincut theorem, as well as the integral flow theorem. The Ford-Fulkerson algorithm is a greedy algorithm: we find a path from s to t of positive capacity and we push as much flow as we can on it (saturating at least one edge on the path). We then describe the capacities left over in a "residual graph" and repeat the process, continuing until there are no more paths of positive residual capacity left between s and t. Remember, one of the key but subtle points here is how we define the residual graph: if we push f units of flow on an edge (u, v), then the residual capacity of (u, v) goes down by f but also the residual capacity of (v, u) goes *up* by f (since pushing flow in the opposite direction is the same as reducing the flow in the forward direction). We then proved that this in fact finds the maximum flow.

Assuming capacities are integers, the basic Ford-Fulkerson algorithm could make up to F iterations, where F is the value of the maximum flow. Each iteration takes $O(m)$ time to find a path using DFS or BFS and to compute the residual graph. (To reduce notation, let's assume we have pre-processed the graph to delete any disconnected parts so that $m \geq n-1$.) So, the overall total time is $O(mF)$.

This is fine if F is small, like in the case of bipartite matching (where $F \leq n$). However, it's not good if capacities are in binary and F could be very large. In fact, it's not hard to construct an example where a series of bad choices of which path to augment on could make the algorithm take a very long time: see Figure 17.1.

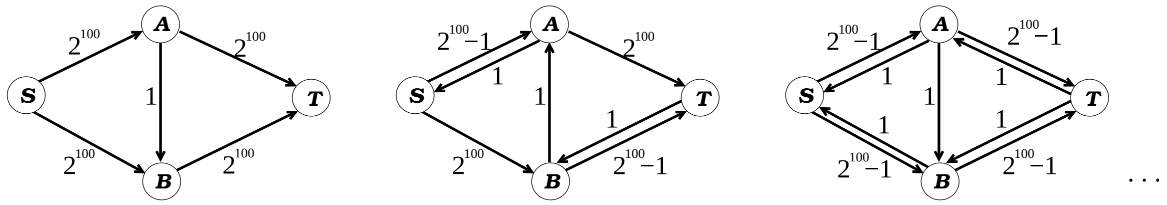


Figure 17.1: A bad case for Ford-Fulkerson. Starting with the graph on the left, we choose the path s-a-b-t, producing the residual graph shown in the middle. We then choose the path s-b-a-t, producing the residual graph on the right, and so on.

Can anyone think of some ideas on how we could speed up the algorithm? Here are two we can prove something about.

17.3 Edmonds-Karp #1

Edmonds-Karp #1 is probably the most natural idea that one could think of. Instead of picking an *arbitrary* path in the residual graph, let's pick the one of largest capacity (we called this the "maximum bottleneck path" a few lectures ago).

Claim 17.1 *In a graph with maximum s-t flow F, there must exist a path from s to t with capacity at least F/m.*

Can anyone think of a proof?

Proof: Suppose we delete all edges of capacity less than F/m. This can't disconnect t from s since if it did we would have produced a cut of value less than F. So, the graph left over must have a path from s to t, and since all edges on it have capacity at least F/m, the path itself has capacity at least F/m. ■

Claim 17.2 *Edmonds-Karp #1 makes at most $O(m \log F)$ iterations.*

Proof: By Claim 17.1, each iteration adds least a $1/m$ fraction of the "flow still to go" (the maximum flow in the current residual graph) to the flow found so far. Or, equivalently, after each iteration, the "flow still to go" gets reduced by a $(1-1/m)$ factor. So, the question about number of iterations just boils down to: given some number F, how many times can you remove a

$1/m$ fraction of the amount remaining until you get down below 1 (which means you are at zero since everything is integral)? Mathematically, for what number x do we have $F(1-1/m)^x < 1$? Notice that $(1-1/m)^m$ is approximately (and always less than) $1/e$. So, $x = m \ln F$ is sufficient: $F(1-1/m)^x < F(1/e)^{\ln F} = 1$. ■

Now, remember we can find the maximum bottleneck path in time $O(m \log n)$, so the overall time used is $O(m^2 \log n \log F)$. You can actually get rid of the "$\log n$" by being a little tricky, bringing this down to $O(m^2 \log F)$.[1]

So, using this strategy, the dependence on F has gone from linear to logarithmic. In particular, this means that even if edge capacities are large integers written in binary, running time is polynomial in the number of bits in the description size of the input.

We might ask, though, can we remove dependence on F completely? It turns out we *can*, using the second Edmonds-Karp algorithm.

17.4 Edmonds-Karp #2

The Edmonds-Karp #2 algorithm works by always picking the *shortest* path in the residual graph (the one with the fewest number of edges), rather than the path of maximum capacity. This sounds a little funny but the claim is that by doing so, the algorithm makes at most mn iterations. So, the running time is $O(nm^2)$ since we can use BFS in each iteration. The proof is pretty neat too.

Claim 17.3 *Edmonds-Karp #2 makes at most mn iterations.*

Proof: Let d be the distance from s to t in the current residual graph. We'll prove the result by showing that (a) d never decreases, and (b) every m iterations, d has to increase by at least 1 (which can happen at most n times).

Let's lay out G in levels according to a BFS from s. That is, nodes at level i are distance i away from s, and t is at level d. Now, keeping this layout fixed, let us observe the sequence of paths found and residual graphs produced. Notice that so long as the paths found use only forward edges in this layout, each iteration will cause at least one forward edge to be saturated and removed from the residual graph, and it will add only backward edges. This means first of all that d does not decrease, and secondly that so long as d has not changed (so the paths *do* use only forward edges), at least one forward edge in this layout gets removed. We can remove forward edges at most m times, so within m iterations either t becomes disconnected (and $d = \infty$) or else we must have used a non-forward edge, implying that d has gone up by 1. We can then re-layout the current residual graph and apply the same argument again, showing that the distance between s and t never decreases, and there can be a gap of size at most m between successive increases.

[1]This works as follows. First, let's find the largest power of 2 (let's call it $c = 2^i$) such that there exists a path from s to t in G of capacity at least c. We can do this in time $O(m \log F)$ by guessing and doubling (starting with $c = 1$, throw out all edges of capacity less than c, and use DFS to check if there is a path from s to t; if a path exists, then double the value of c and repeat). Now, instead of looking for *maximum* capacity paths in the Edmonds-Karp algorithm, we just look for s-t paths of residual capacity $\geq c$. The advantage of this is we can do this in linear time with DFS. If no such path exists, divide c by 2 and try again. The result is we are always finding a path that is within a factor of 2 of having the maximum capacity (so the bound in Claim 17.2 still holds), but now it only takes us $O(m)$ time per iteration rather than $O(m \log n)$.

Since the distance between s and t can increase at most n times, this implies that in total we have at most nm iterations. ■

17.5 Further discussion: Dinic and MPM

Can we do this a little faster? (We may skip this depending on time, and in any case the details here are not so important for this course, but the high level idea is nice.)

The previous algorithm used $O(mn)$ iterations at $O(m)$ time each for $O(m^2n)$ time total. We'll now see how we can reduce to $O(mn^2)$ time and finally to time $O(n^3)$. Here is the idea: given the current BFS layout used in the Edmonds-Karp argument (also called the "level graph"), we'll try in $O(n^2)$ time all at once to find the maximum flow that only uses forward edges. This is sometimes called a "blocking flow". Just as in the analysis above, such a flow guarantees that when we take the residual graph, there is no longer an augmenting path using only forward edges and so the distance to t will have gone up by 1. So, there will be at most n iterations for a total time of $O(n^3)$.

To describe the algorithm, define the *capacity of a vertex* v as $c(v) = \min[c_{in}(v), c_{out}(v)]$, where $c_{in}(v)$ is the sum of capacities of the in-edges to v and $c_{out}(v)$ is the sum of capacities of the out-edges from v. The algorithm is now as follows:

1. In $O(m)$ time, create the level graph G_{level} which is a BFS layout from s containing only the forward edges, where we then remove any nodes that can't reach t using these edges (e.g., by doing a backwards BFS from t and removing unmarked nodes). Compute the capacities of all nodes, considering just edges in G_{level}.

2. Find the vertex v of minimum capacity c. If it's zero, that's great: we can remove v and incident edges, updating capacities of neighboring nodes.

3. If c is not zero, then we greedily pull c units of flow from s to v, and then push that flow along to t. We then update the capacities, delete v and repeat. Unfortunately, this seems like just another version of the original problem! But, there are two points here:

   (a) Because v had *minimum* capacity, we can do the pulling and pushing in any greedy way and we won't get stuck. E.g., we can do a BFS backward from v, pulling the flow level-by-level, so we never examine any edge twice (do an example here), and then a separate BFS forward from v doing the same thing to push the c units forward to t.

   This right away gives us an $O(mn)$ algorithm for saturating the level graph ($O(m)$ per node, n nodes), for an overall running time of $O(mn^2)$. So, we're half-way there.

   (b) To improve the running time further, the way we will do the BFS is to examine the in-edges (or out-edges) one at a time, fully saturating the edge before going on to the next one. This means we can allocate our time into two parts: (a) time spent pushing/pulling through edges that get saturated, and (b) time spent on edges that we didn't quite saturate (at most one of these per node). In the BFS we only take time $O(n)$ on type-b operations since we do at most one of these per vertex. We may spend more time on type-a operations, but those result in deleting the edge, so the edge won't be used again in the current level graph. So over *all* vertices v used in this process, the *total* time of these type-a operations is $O(m)$. That means we can saturate the level graph in time $O(n^2 + m) = O(n^2)$.

Our running time to saturate the level graph is $O(n^2 + m) = O(n^2)$. Once we've saturated the level graph, we recompute it (in $O(m)$ time), and re-solve. The total time is $O(n^3)$.

By the way, even this is not the best running time known: the currently best algorithms have performance of almost $O(mn)$.

17.6 Min-cost Matchings, Min-cost Max Flows

We talked about the problem of assigning groups to time-slots where each group had a list of acceptable versus unacceptable slots. A natural generalization is to ask: what about preferences? E.g, maybe group A prefers slot 1 so it costs only \$1 to match to there, their second choice is slot 2 so it costs us \$2 to match the group here, and it can't make slot 3 so it costs \$infinity to match the group to there. And, so on with the other groups. Then we could ask for the *minimum cost* perfect matching. This is a perfect matching that, out of all perfect matchings, has the least total cost.

The generalization of this problem to flows is called the *min-cost max flow* problem. Formally, the min-cost max flow problem is defined as follows. We are given a graph G where each edge has a *cost* $w(e)$ in as well as a capacity $c(e)$. The cost of a flow is the sum over all edges of the positive flow on that edge times the cost of the edge. That is,

$$cost(f) = \sum_e w(e) f^+(e),$$

where $f^+(e) = \max[f(e), 0]$. Our goal is to find, out of all possible maximum flows, the one with the least total cost. We can have negative costs (or benefits) on edges too, but let's assume just for simplicity that the graph has no negative-cost cycles. (Otherwise, the min-cost max flow will have little disconnected cycles in it; one can solve the problem without this assumption, but it's conceptually easier with it.)

Min-cost max flow is more general than plain max flow so it can model more things. For example, it can model the min-cost matching problem described above.

There are several ways to solve the min-cost max-flow problem. One way is we can run Ford-Fulkerson, where each time we choose the *least cost* path from s to t. In other words, we find the shortest path but using the costs as distances. To do this correctly, when we add a back-edge to some edge e into the residual graph, we give it a cost of $-w(e)$, representing that we get our money back if we undo the flow on it. So, this procedure will create residual graphs with negative-weight edges, but we can still find shortest paths in them using the Bellman-Ford algorithm.

We can argue correctness for this procedure as follows. Remember that we assumed the initial graph G had no negative-cost cycles. Now, even though each augmenting path might add new negative-cost *edges*, it will never create a negative-cost *cycle* in the residual graph. That is because if it did, say containing some back-edge (v, u), that means the path using (u, v) wasn't really shortest since a better way to get from u to v would have been to travel around the cycle instead. So, this implies that the final flow f found has the property that G_f has no negative-cost cycles either. This means that f is optimal: if g was a maximum flow of lower cost, then $g - f$ is a legal circulation in G_f (a flow satisfying flow-in = flow-out at *all* nodes including s and t) with negative cost, which means it would have to contain a negative-cost cycle, a contradiction.

The running time for this algorithm is similar to Ford-Fulkerson, except using Bellman-Ford instead of Dijkstra. It is possible to speed it up, but we won't discuss that in this course.

Lecture 18

Linear Programming

18.1 Overview

In this lecture we describe a very general problem called *linear programming* that can be used to express a wide variety of different kinds of problems. We can use algorithms for linear programming to solve the max-flow problem, solve the min-cost max-flow problem, find minimax-optimal strategies in games, and many other things. We will primarily discuss the setting and how to code up various problems as linear programs At the end, we will briefly describe some of the algorithms for solving linear programming problems. Specific topics include:

- The definition of linear programming and simple examples.
- Using linear programming to solve max flow and min-cost max flow.
- Using linear programming to solve for minimax-optimal strategies in games.
- Algorithms for linear programming.

18.2 Introduction

In the last two lectures we looked at:

— Bipartite matching: given a bipartite graph, find the largest set of edges with no endpoints in common.

— Network flow (more general than bipartite matching).

— Min-Cost Max-flow (even more general than plain max flow).

Today, we'll look at something even more general that we can solve algorithmically: **linear programming**. (Except we won't necessarily be able to get integer solutions, even when the specification of the problem is integral).

Linear Programming is important because it is so expressive: many, *many* problems can be coded up as linear programs (LPs). This especially includes problems of allocating resources and business supply-chain applications. In business schools and Operations Research departments there are entire courses devoted to linear programming. We're only going to have time for 1 lecture. So, we will just have time to say what they are, and give examples of encoding problems as LPs. We will only say a tiny bit about algorithms for solving them.

Before defining the problem, let's motivate it with an example:

Example: There are 168 hours in a week. Say we want to allocate our time between classes and studying (S), fun activities and going to parties (P), and everything else (E) (eating, sleeping, taking showers, etc). Suppose that to survive we need to spend at least 56 hours on E (8 hours/day). To maintain sanity we need $P + E \geq 70$. To pass our courses, we need $S \geq 60$, but more if don't sleep enough or spend too much time partying: $2S + E - 3P \geq 150$. (E.g., if don't go to parties at all then this isn't a problem, but if we spend more time on P then need to sleep more or study more).

Q1: Can we do this? Formally, is there a *feasible* solution?

A: Yes. For instance, one feasible solution is: $S = 80, P = 20, E = 68$.

Q2: Suppose our notion of happiness is expressed by $2P + E$. What is a feasible solution such that this is maximized? The formula "$2P + E$" is called an *objective function*.

The above is an example of a *linear program*. What makes it linear is that all our constraints are linear inequalities in our variables. E.g., $2S + E - 3P \geq 150$. In addition, our objective function is also linear. We're not allowed things like requiring $SE \geq 100$, since this wouldn't be a linear inequality.

18.3 Definition of Linear Programming

More formally, a linear programming problem is specified as follows.

Given:

- n variables $x_1, \ldots, x_n$.
- m linear inequalities in these variables (equalities OK too).

  E.g., $3x_1 + 4x_2 \leq 6$, $0 \leq x_1 \leq 3$, etc.
- We may also have a linear objective function. E.g., $2x_1 + 3x_2 + x_3$.

Goal:

- Find values for the x_i's that satisfy the constraints and maximize the objective. (In the "feasibility problem" there is no objective function: we just want to satisfy the constraints.)

For instance, let's write out our time allocation problem this way.

Variables: S, P, E.

Objective: maximize $2P + E$, subject to

Constraints:

$$\begin{aligned} S+P+E &= 168 \\ E &\geq 56 \\ S &\geq 60 \\ 2S+E-3P &\geq 150 \\ P+E &\geq 70 \\ P &\geq 0 \qquad \text{(can't spend negative time partying)} \end{aligned}$$

18.4 Modeling problems as Linear Programs

Here is a typical Operations-Research kind of problem (stolen from Mike Trick's course notes): Suppose you have 4 production plants for making cars. Each works a little differently in terms of labor needed, materials, and pollution produced per car:

	labor	materials	pollution
plant 1	2	3	15
plant 2	3	4	10
plant 3	4	5	9
plant 4	5	6	7

Suppose we need to produce at least 400 cars at plant 3 according to a labor agreement. We have 3300 hours of labor and 4000 units of material available. We are allowed to produce 12000 units of pollution, and we want to maximize the number of cars produced. How can we model this?

To model a problem like this, it helps to ask the following three questions in order: (1) what are the variables, (2) what is our objective in terms of these variables, and (3) what are the constaints. Let's go through these questions for this problem.

1. What are the variables? x_1, x_2, x_3, x_4, where x_i denotes the number of cars at plant i.
2. What is our objective? maximize $x_1 + x_2 + x_3 + x_4$.
3. What are the constraints?

$$\begin{aligned} x_i &\geq 0 \quad \text{(for all } i\text{)} \\ x_3 &\geq 400 \\ 2x_1+3x_2+4x_3+5x_4 &\leq 3300 \\ 3x_1+4x_2+5x_3+6x_4 &\leq 4000 \\ 15x_1+10x_2+9x_3+7x_4 &\leq 12000 \end{aligned}$$

Note that we are not guaranteed the solution produced by linear programming will be integral. For problems where the numbers we are solving for are large (like here), it is usually not a very big deal because you can just round them down to get an almost-optimal solution. However, we will see problems later where it *is* a very big deal.

18.5 Modeling Network Flow

We can model the max flow problem as a linear program too.

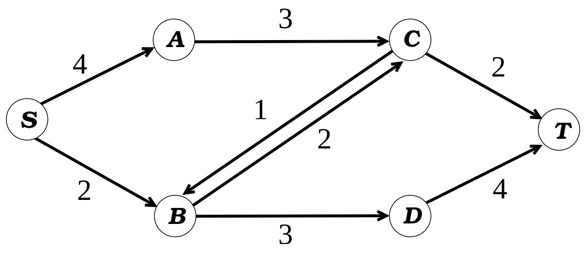
Variables: Set up one variable x_{uv} for each edge (u, v). Let's just represent the positive flow since it will be a little easier with fewer constraints.

Objective: Maximize $\sum_u x_{ut} - \sum_u x_{tu}$. (maximize the flow into t minus any flow out of t)

Constraints:

- For all edges (u, v), $0 \leq x_{uv} \leq c(u, v)$. (capacity constraints)
- For all $v \notin \{s, t\}$, $\sum_u x_{uv} = \sum_u x_{vu}$. (flow conservation)

For instance, consider the example from the network-flow lecture:



In this case, our LP is: maximize $x_{ct} + x_{dt}$ subject to the constraints:

$0 \leq x_{sa} \leq 4$, $0 \leq x_{ac} \leq 3$, etc.

$x_{sa} = x_{ac}$, $x_{sb} + x_{cb} = x_{bc} + x_{bd}$, $x_{ac} + x_{bc} = x_{cb} + x_{ct}$, $x_{bd} = x_{dt}$.

How about min cost max flow? We can do this in two different ways. One way is to first solve for the maximum flow f, ignoring costs. Then, add a *constraint* that flow must equal f, and subject to that constraint (plus the original capacity and flow conservation constraints), minimize the linear cost function

$$\sum_{(u,v)\in E} w(u, v)x_{uv},$$

where $w(u, v)$ is the cost of edge (u, v). Alternatively, you can solve this all in one step by adding an edge of infinite capacity and very negative cost from t to s, and then just minimizing cost (which will automatically maximize flow).

18.6 2-Player Zero-Sum Games

Recall back to homework 1, we looked at a 2-player zero-sum game. We had Alice hiding a nickel or quarter behind her back and Bob guessing, and then based on whether the guess was right or wrong, some money changed hands. This is called a "zero-sum game" because no money is entering or leaving the system (it's all going between Alice and Bob).

A **Minimax Optimal** strategy for a player is a (possibly randomized) strategy with the best guarantee on its expected gain over strategies the opponent could play in response — i.e., it is the strategy you would want to play if you imagine that your opponent knows you well.

Here is another game: Suppose a kicker is shooting a penalty kick against a goalie who is a bit weaker on one side. Let's say the kicker can kick left or right, the goalie can dive left or right, and the payoff matrix for the kicker (the chance of getting a goal) looks as follows:

Goalie

Kicker

	left	*right*
left	0	1
right	1	0.5

Notice that even though the goalie is weaker on the right, the minimax optimal strategy for the kicker is not to always kick to the right since a goalie who knows the kicker well could then stop half the shots. Instead, the minimax optimal strategy for the kicker is to kick right with probability 2/3 and to kick left with probability 1/3. This guarantees an expected gain of 2/3 no matter how the goalie decides to dive.

How about solving an n by n game? For instance, perhaps a game like this:

$$\begin{array}{|rrr|}
\hline
20 & -10 & 5 \\
5 & 10 & -10 \\
-5 & 0 & 10 \\
\hline
\end{array}$$

Let's see if we can use linear programming to compute a minimax optimal strategy (say, for the row player). Assume we are given as input an array M where m_{ij} represents the payoff to the row player when the row player plays i and the column player plays j.

Informally, we want the variables to be the things we want to figure out, which in this case are the probabilities to put on our different choices $p_1, \ldots, p_n$. These have to form a legal probability distribution, and we can describe this using linear inequalities: namely, $p_1 + \ldots + p_n = 1$ and $p_i \geq 0$ for all i.

Our goal is to maximize the worst case (minimum), over all columns our opponent can play, of our expected gain. This is a little confusing because we are maximizing a minimum. However, we can use a trick: we will add one new variable v (representing the minimum), put in *constraints* that our expected gain has to be at least v for every column, and then define our objective to be to maximize v. Putting this all together we have:

Variables: $p_1, \ldots, p_n$ and v.

Objective: Maximize v.

Constraints:

- $p_i \geq 0$ for all i, and $\sum_i p_i = 1$. (the p_i form a probability distribution)
- for all columns j, we have $\sum_i p_i m_{ij} \geq v$.

18.7 Algorithms for Linear Programming

How can we solve linear programs? The standard algorithm for solving LPs is the Simplex Algorithm, developed in the 1940s. It's *not* guaranteed to run in polynomial time, and you *can* come up with bad examples for it, but in general the algorithm runs pretty fast. Only much later in 1980 was it shown that linear programming could be done in polynomial time by something called the Ellipsoid Algorithm (but it tends to be fairly slow in practice). Later on, a faster polynomial-time algorithm called Karmarkar's Algorithm was developed, which is competitive with Simplex. There are many commercial LP packages, for instance LINDO, CPLEX, Solver (in Excel) and others.

We won't have time to describe any of these algorithms in detail. Instead, we will just give some intuition and the high-level idea of how they work by viewing linear programming as a geometrical problem.

Think of an n-dimensional space with one coordinate per variable. A solution is a point in this space. An inequality, like $x_1 + x_2 \leq 6$ is saying that we need the solution to be on a specified side of a certain hyperplane. The feasible region is the convex region in space defined by these constraints. Then we want to find the feasible point that is farthest in the "objective" direction.

Let's go back to our first example with S, P, and E. To make this easier to draw, we can use our first constraint that $S + P + E = 168$ to replace S with $168 - P - E$. This means we can just draw in 2 dimensions, P and E. See Figure 18.1.

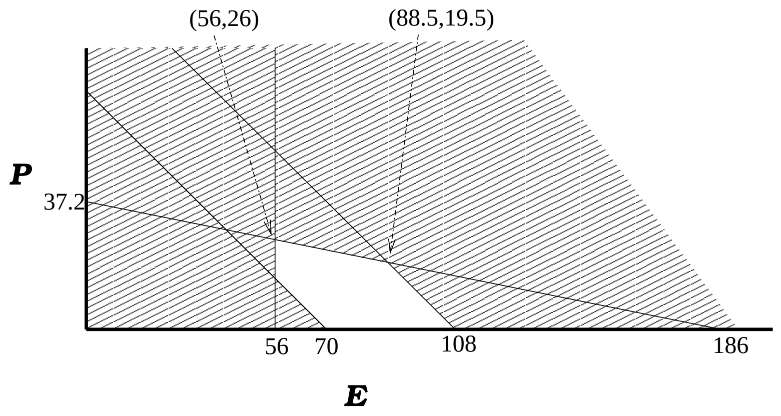


Figure 18.1: Feasible region for our time-planning problem. The constraints are: $E \geq 56$; $P + E \geq 70$; $P \geq 0$; $S \geq 60$ which means $168 - P - E \geq 60$ or $P + E \leq 108$; and finally $2S - 3P + E \geq 150$ which means $2(168 - P - E) - 3P + E \geq 150$ or $5P + E \leq 186$.

We can see from the figure that for the objective of maximizing P, the optimum happens at $E = 56, P = 26$. For the objective of maximizing $2P + E$, the optimum happens at $E = 88.5, P = 19.5$.

We can use this geometric view to motivate the algorithms.

The Simplex Algorithm: The earliest and most common algorithm in use is called the Simplex method. The idea is to start at some "corner" of the feasible region (to make this easier, we can add in so-called "slack variables" that will drop out when we do our optimization). Then we repeatedly do the following step: look at all neighboring corners of our current position and go to the best one (the one for which the objective function is greatest) if it is better than our current position. Stop when we get to a corner where no neighbor has a higher objective value than we currently have.

The key fact here is that (a) since the objective is *linear*, the optimal solution will be at a corner (or maybe multiple corners). Furthermore, (b) there are no local maxima: if you're *not* optimal, then some neighbor of you must have a strictly larger objective value than you have. That's because the feasible region is *convex*. So, the Simplex method is guaranteed to halt at the best solution. The problem is that it is possible for there to be an exponential number of corners and it is possible for Simplex to take an exponential number of steps to converge. But, in practice this usually works well.

The Ellipsoid Algorithm: The Ellipsoid Algorithm was invented by Khachiyan in 1980 in Russia.

This algorithms solves just the "feasibility problem," but you can then do binary search with the objective function to solve the optimization problem. The idea is to start with a big ellipse (called an ellipsoid in higher dimensions) that we can be sure contains the feasible region. Then, try the center of the ellipse to see if it violates any constraints. If not, you're done. If it does, then look at some constraint violated. So we know the solution (if any) is contained in the remaining at-most-half-ellipse. Now, find a new smaller ellipse that contains that half of our initial ellipse. We then repeat with the new smaller ellipse. One can show that in each step, you can always create a new smaller ellipse whose volume is smaller, by at least a $(1 - 1/n)$ factor, than the original ellipse. So, every n steps, the volume has dropped by about a factor of $1/e$. One can then show that if you ever get *too* small a volume, as a function of the number of bits used in the coefficients of the constraints, then that means there is no solution after all.

One nice thing about the Ellipsoid Algorithm is you just need to tell if the current solution violates any constraints or not, and if so, to produce one. You don't need to explicitly write them all down. There are some problems that you can write as a linear program with an exponential number of constraints if you had to write them down explicitly, but where there is an fast algorithm to determine if a proposed solution violates any constraints and if so to produce one. For these kinds of problems, the Ellipsoid Algorithm is a good one.

Karmarkar's Algorithm: Karmarkar's Algorithms sort of has aspects of both. It works with feasible points but doesn't go from corner to corner. Instead it moves inside the interior of the feasible region. It was one of first of a whole class of so-called "interior-point" methods.

The development of better and better algorithms is a big ongoing area of research. In practice, for all of these algorithms, you get a lot of mileage by using good data structures to speed up the time needed for making each decision.

Lecture 19

NP-Completeness I

19.1 Overview

In the past few lectures we have looked at increasingly more expressive problems that we were able to solve using efficient algorithms. In this lecture we introduce a class of problems that are so expressive — they are able to model *any* problem in an extremely large class called **NP** — that we believe them to be *intrinsically unsolvable by polynomial-time algorithms.* These are the **NP-complete** problems. What is particularly surprising about this class is that they include many problems that at first glance appear to be quite benign. Specific topics in this lecture include:

- Reductions and expressiveness
- Informal definitions and the ESP problem
- Formal definitions: decision problems, P and NP.
- Circuit-SAT and 3-SAT

19.2 Introduction: Reduction and Expressiveness

In the last few lectures have seen a series of increasingly more expressive problems: network flow, min cost max flow, and finally linear programming. These problems have the property that you can code up a lot of different problems in their "language". So, by solving these well, we end up with important tools we can use to solve other problems.

To talk about this a little more precisely, it is helpful to make the following definitions:

Definition 19.1 *We say that an algorithm runs in* **Polynomial Time** *if, for some constant c, its running time is $O(n^c)$, where n is the size of the input.*

In the above definition, "size of input" means "number of bits it takes to write the input down". So, to be precise, when defining a problem and asking whether or not a certain algorithm runs in polynomial time, it is important to say how the input is given. For instance, the basic Ford-Fulkerson algorithm is *not* a polynomial-time algorithm for network flow when edge capacities are written in binary, but both of the Edmonds-Karp algorithms *are* polynomial-time.

Definition 19.2 *A Problem A is* **poly-time reducible** *to problem B (written as $A \leq_p B$) if we can solve problem A in polynomial time given a polynomial time black-box algorithm for problem B. Problem A is* **poly-time equivalent** *to problem B ($A =_p B$) if $A \leq_p B$ and $B \leq_p A$.*

For instance, we gave an efficient algorithm for Bipartite Matching by showing it was poly-time reducible to Max Flow. Notice that it could be that $A \leq_p B$ and yet our fastest algorithm for solving problem A might be slower than our fastest algorithm for solving problem B (because our reduction might involve several calls to the algorithm for problem B, or might involve blowing up the input size by a polynomial but still nontrivial amount).

19.3 Our first NP-Complete Problem: ESP

Many of the problems we would like to solve have the property that if someone handed us a solution, we could at least check if the solution was correct. For instance the Traveling Salesman Problem asks: "Given a weighted graph G and an integer k, does G have a tour that visits all the vertices and has total length at most k?" We may not know how to find such a tour quickly, but if someone gave such a tour to us, we could easily check if it satisfied the desired conditions (visited all the vertices and had total length at most k). Similarly, for the 3-Coloring problem: "Given a graph G, can vertices be assigned colors red, blue, and green so that no two neighbors have the same color?" we don't know of any polynomial-time algorithms for solving the problem but we could easily check a proposed solution if someone gave one to us. The class of problems of this type — namely, if the answer is YES, then there exists a polynomial-length proof that can be checked in polynomial time — is called **NP**. (we define the class **NP** formally in Section 19.5).

Let's consider now what would be a problem *so expressive* that if we could solve it, we could solve any problem of this kind. Moreover, let's see if we can define the problem so that it is of this kind as well. Here is a natural candidate:

Definition 19.3 Existence of a verifiable Solution Problem (ESP)*: The input to this problem is in three parts. The first part is a program $V(I, X)$, written in some standard programming language, that has two arguments.*[1] *The second part is a string I intended as a first argument, and the third part is a bound b written in unary (a string of b 1s). Question: does there exist a string X, $|X| \leq b$, such that $V(I, X)$ halts in at most b steps and outputs YES?*

What we will show is that (a) ESP $\in$ **NP** and (b) for any problem $Q \in$ **NP** we have $Q \leq_p$ ESP. (I.e., if you "had ESP" you could solve any problem in **NP**).[2]

Let's begin with (a): why is ESP $\in$ **NP**? This is the reason for the bound b written in unary. If we didn't have b at all, then (since we can't even in general tell if a program is ever going to halt) the ESP question would not even be computable. However, with the bound b, if the answer is YES, then there is a short proof (namely the string X) that we can check in polynomial time (just run $V(I, X)$ for b steps). The reason we ask for b to be written in unary is precisely so that this check counts as being polynomial time: if b were in binary, then this check could take time exponential

[1]We use "V" for the program because we will think of it as a solution-verifier.

[2]Thanks to Manuel Blum for suggesting the acronym.

in the number of bits in the input (much like Ford-Fulkerson is not a polynomial-time algorithm if the capacities are written in binary).

Now, let's go to (b): why is it the case that for any problem $Q \in \mathbf{NP}$ we have $Q \leq_p$ ESP? Consider some **NP** problem we might want to solve like 3-Coloring. We don't know any fast ways of solving that problem, but we can easily write a program V that given inputs $I = G$ and $X =$ an assignment of colors to the vertices, verifies whether X indeed satisfies our requirements (uses at most three colors and gives no two adjacent vertices the same color). Furthermore, this solution-verifier is linear time. So, if we had an algorithm to solve the ESP, we could feed in this V, feed in the graph G, feed in a bound b that is linear in the size of G, and solve the 3-Coloring problem. Similarly, we could do this for the Traveling Salesman Problem: program V, given inputs $I = (G, k)$ and $X =$ a description of a tour through G, just verifies that the tour indeed has length at most k and visits all the vertices. More generally, we can do this for any problem Q in **NP**. By definition of **NP**, YES-instances of Q must have short proofs that can be easily checked: i.e., they must have such a solution-verifier V that we can plug into our magic ESP algorithm.

Thus, we have shown that ESP satisfies both conditions (a) and (b) and therefore is **NP-complete**.

19.4 Search versus Decision

Technically, a polynomial-time algorithm for the ESP just tells us if a solution exists, but doesn't actually produce it. How could we use an algorithm that just answers the YES/NO question of ESP to actually find a solution X? If we can do this, then we can use it to actually *find* the coloring or *find* the tour, not just smugly tell us that there is one. The problem of actually finding a solution is often called the *search* version of the problem, as opposed to the *decision* version that just asks whether or not the solution exists. That is, we are asking: can we reduce the search version of the ESP to the decision version?

It turns out that in fact we can, by essentially performing binary search. In particular, once we know that a solution X exists, we want to ask: "how about a solution whose first bit is 0?" If, say, the answer to that is YES, then we will ask: "how about a solution whose first two bits are 00?" If, say, the answer to that is NO (so there must exist a solution whose first two bits are 01) we will then ask: "how about a solution whose first three bits are 010?" And so on. The key point is that we can do this using a black-box algorithm for the decision version of ESP as follows. Given a string of bits S, we define a new program $V_S(I, X) = V(I, X_S)$ where X_S is the string X whose first $|S|$ bits are replaced by S. We then feed our magic ESP algorithm the program V_S instead of V. This way, using at most b calls to the decision algorithm, we can solve the search problem too.

So, if we had a polynomial-time algorithm for the decision version of ESP, we immediately get a polynomial-time algorithm for the search version of ESP. So, we can *find* the tour or coloring or whatever.

The ESP seems pretty stylized. But we can now show that other simpler-looking problems have the property that if you could solve them in polynomial-time, then you could solve the ESP in polynomial time as well, so they too are **NP**-complete. That is, they are so expressive that if we *could* solve them in polynomial-time, then it would mean that for any problem where we could *check* a proposed solution efficiently, we could also *find* such a solution efficiently. Now, onto formal definitions.

19.5 Formal definitions: P, NP, and NP-Completeness

We will formally be considering decision problems: problems whose answer is YES or NO. E.g., "Does the given network have a flow of value at least k?" or "Does the given graph have a 3-coloring?" For such problems, we can split all possible instances into two categories: YES-instances (whose correct answer is YES) and NO-instances (whose correct answer is NO). We can also put any ill-formed instances into the NO category. We now define the complexity classes **P** and **NP**.

Definition 19.4 **P** *is the set of decision problems solvable in polynomial time.*

E.g., the decision version of the network flow problem: "Given a network G and a flow value k, does there exist a flow $\geq k$?" belongs to **P**.

Definition 19.5 **NP** *is the set of decision problems that have polynomial-time* verifiers. *Specificially, problem Q is in* **NP** *if there is a polynomial-time algorithm $V(I, X)$ such that:*

- *If I is a YES-instance, then there exists X such that $V(I, X) =$ YES.*
- *If I is a NO-instance, then for all X, $V(I, X) =$ NO.*

Furthermore, X should have length polynomial in size of I (since we are really only giving V time polynomial in the size of the instance, not the combined size of the instance and solution).

The second input X to the verifier V is often called a *witness*. E.g., for 3-coloring, the witness that an answer is YES is the coloring. For factoring, the witness that N has a factor between 2 and k is a factor. For the TRAVELING SALESMAN PROBLEM: "Given a weighted graph G and an integer k, does G have a tour that visits all the vertices and has total length at most k?" the witness is the tour. All these problems belong to **NP**. Of course, any problem in **P** is also in **NP**, since V could just ignore X and directly solve I. So, $\mathbf{P} \subseteq \mathbf{NP}$.

A huge open question in complexity theory is whether $\mathbf{P} = \mathbf{NP}$. It would be quite strange if they were equal since that would mean that any problem for which a solution can be easily *verified* also has the property that a solution can be easily *found*. So most people believe $\mathbf{P} \neq \mathbf{NP}$. But, it's very hard to prove that a fast algorithm for something does *not* exist. So, it's still an open problem.

Definition 19.6 *Problem Q is* **NP***-complete if:*

1. *Q is in* **NP***, and*
2. *For any other problem Q' in* **NP***, $Q' \leq_p Q$.*

So if Q is **NP**-complete and you could solve Q in polynomial time, you could solve *any* problem in **NP** in polynomial time. If Q just satisfies part (2) of the definition, then it's called **NP**-hard.

As we showed above, the ESP is **NP**-complete: it belongs to **NP** (that was the reason for including the bound b in unary, so that running the verifier for b steps counts as being polynomial-time), and we saw that we could use a polynomial-time algorithm for the ESP to solve any other problem in **NP** in polynomial-time.

19.6 Circuit-SAT and 3-SAT

Though the ESP is **NP**-complete, it is a bit unweildy. We will now develop two more natural problems that also are **NP**-complete: CIRCUIT-SAT and 3-SAT. Both of them will be obviously in **NP**. To show they are **NP**-complete, we will show that ESP $\leq_p$ CIRCUIT-SAT, and then that CIRCUIT-SAT $\leq_p$ 3-SAT. Notice that this is enough: it means that if you had a polynomial-time algorithm for 3-SAT then you would also have a polynomial-time algorithm for CIRCUIT-SAT; and if you had a polynomial-time algorithm for CIRCUIT-SAT, then you would also have a polynomial-time algorithm for ESP; and we already know that if you have a polynomial-time algorithm for ESP, you can solve any problem in **NP** in polynomial-time. In other words, to show that a problem Q is **NP**-complete, we just need to show that $Q' \leq_p Q$ for *some* **NP**-complete problem Q' (plus show that $Q \in \mathbf{NP}$).

Definition 19.7 CIRCUIT-SAT: *Given a circuit of NAND gates with a single output and no loops (some of the inputs may be hardwired). Question: is there a setting of the inputs that causes the circuit to output 1?*

Theorem 19.1 *CIRCUIT-SAT is* **NP***-complete.*

Proof Sketch: First of all, CIRCUIT-SAT is clearly in **NP**, since you can just guess the input and try it. To show it is **NP**-complete, we need to show that if we could solve this, then we could solve the ESP. Say we are given V, I, and b, and want to tell if there exists X such that $V(I, X)$ halts and outputs YES within at most than b steps. Since we only care about running V for b steps we can assume it uses at most b bits of memory, including the space for its arguments. We will now use the fact that one can construct a RAM with b bits of memory (including its stored program) and a standard instruction set using only $O(b \log b)$ NAND gates and a clock. By unrolling this design for b levels, we can remove loops and create a circuit that simulates what V computes within b time steps. We then hardwire the inputs corresponding to I and feed this into our CIRCUIT-SAT solver. ■

So, we now have one more **NP**-complete problem. Still, CIRCUIT-SAT looks complicated: we weren't expecting to be able to solve it in polynomial-time. However, now we will show that a much simpler-looking problem, 3-SAT has the property that CIRCUIT-SAT $\leq_p$ 3-SAT.

Definition 19.8 3-SAT: *Given: a CNF formula (AND of ORs) over* n *variables* $x_1, \ldots, x_n$*, where each clause has at most 3 variables in it. E.g.,* $(x_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_2 \vee x_3) \wedge (x_1 \vee x_3) \wedge \ldots$*. Goal: find an assignment to the variables that satisfies the formula if one exists.*

We'll save the proof that 3-SAT is **NP**-complete for the next lecture, but before we end, here is formally how we are going to do our reductions. Say we have some problem A that we know is **NP**-complete. We want to show problem B is **NP**-complete too. Well, first we show B is in **NP** but that is usually the easy part. The main thing we need to do is show that $A \leq_p B$; that is, any polynomial-time algorithm for B would give a polynomial-time algorithm for A. We will do this through the following method called a *many-one* or *Karp* reduction:

Many-one (Karp) reduction from problem A to problem B: To reduce problem A to problem B we want a function f that takes arbitrary instances of A to instances of B such that:

1. if x is a YES-instance of A then $f(x)$ is a YES-instance of B.
2. if x is a NO-instance of A then $f(x)$ is a NO-instance of B.
3. f can be computed in polynomial time.

So, if we had an algorithm for B, and a function f with the above properties, we could using it to solve A on any instance x by running it on $f(x)$.

Lecture 20

NP-Completeness II

20.1 Overview

In the last lecture, we defined the class **NP** and the notion of **NP**-completeness, and proved that the Circuit-SAT problem is **NP**-complete. In this lecture we continue our discussion of NP-Completeness, showing the following results:

- CIRCUIT-SAT $\leq_p$ 3-SAT (proving 3-SAT is NP-complete)
- 3-SAT $\leq_p$ CLIQUE (proving CLIQUE is NP-complete)
- NP-completeness of Independent Set and Vertex Cover

20.2 Introduction

Let us begin with a quick recap of our discussion in the last lecture. First of all, to be clear in our terminology, a *problem* means something like 3-coloring or network flow, and an *instance* means a specific instance of that problem: the graph to color, or the network and distinguished nodes s and t we want to find the flow between. A *decision problem* is just a problem where each instance is either a YES-instance or a NO-instance, and the goal is to decide which type your given instance is. E.g., for 3-coloring, G is a YES-instance if it has a 3-coloring and is a NO-instance if not. For the Traveling Salesman Problem, an instance consists of a graph G together with an integer k, and the pair (G, k) is a YES-instance iff G has a TSP tour of total length at most k.

We now define our key problem classes of interest.

P: The class of decision problems Q that have polynomial-time algorithms. $Q \in \mathbf{P}$ if there exists a polynomial-time algorithm A such that $A(I) =$ YES iff I is a YES-instance of Q.

NP: The class of decision problems where at least the YES-instances have short proofs (that can be checked in polynomial-time). $Q \in \mathbf{NP}$ if there exists a verifier $V(I, X)$ such that:

- If I is a YES-instance, then there exists X such that $V(I, X) =$ YES,
- If I is a NO-instance, then for all X, $V(I, X) =$ NO,

and furthermore the length of X and the running time of V are polynomial in $|I|$.

co-NP: vice-versa — there are short proofs for NO-instances. Specifically, $Q \in$ **co-NP** if there exists a verifier $V(I, X)$ such that:

- If I is a YES-instance, for all X, $V(I, X) =$ YES,
- If I is a NO-instance, then there exists X such that $V(I, X) =$ NO,

and furthermore the length of X and the running time of V are polynomial in $|I|$.

For example, the problem CIRCUIT-EQUIVALENCE: "Given two circuits C_1, C_2, do they compute the same function?" is in **co-NP**, because if the answer is NO, then there is a short, easily verified proof (an input x such that $C_1(x) \neq C_2(x)$).

A problem Q is **NP**-complete if:

1. $Q \in$ **NP**, and
2. Any other Q' in **NP** is polynomial-time reducible to Q; that is, $Q' \leq_p Q$.

If Q just satisfies (2) then it's called **NP**-hard. Last time we showed that the following problem is **NP**-complete:

> **Circuit-SAT**: Given a circuit of NAND gates with a single output and no loops (some of the inputs may be hardwired). Question: is there a setting of the inputs that causes the circuit to output 1?

Aside: we could define the *search*-version of a problem in **NP** as: "...and furthermore, if I is a YES-instance, then *produce* X such that $V(I, X) =$ YES." If we can solve any **NP**-complete decision problem in polynomial time then we can actually solve search-version of any problem in **NP** in polynomial-time too. The reason is that if we can solve an **NP**-complete problem in polynomial time, then we can solve the ESP in polynomial time, and we already saw how that allows us to produce the X for any given verifier V.

Unfortunately, Circuit-SAT is a little unweildy. What's *especially interesting* about NP-completeness is not just that such problems *exist*, but that a lot of very innocuous-looking problems are NP-complete. To show results of this form, we will first reduce Circuit-SAT to the much simpler-looking 3-SAT problem (i.e., show Circuit-SAT $\leq_p$ 3-SAT). Recall the definition of 3-SAT from last time:

Definition 20.1 3-SAT: *Given: a CNF formula (AND of ORs) over n variables $x_1, \ldots, x_n$, where each clause has at most 3 variables in it. Goal: find an assignment to the variables that satisfies the formula if one exists.*

Theorem 20.1 *CIRCUIT-SAT $\leq_p$ 3-SAT. I.e., if we can solve 3-SAT in polynomial time, then we can solve CIRCUIT-SAT in polynomial time (and thus all of* **NP***).*

Proof: We need to define a function f that converts instances C of Circuit-SAT to instances of 3-SAT such that the formula $f(C)$ produced is satisfiable iff the circuit C had an input x such that $C(x) = 1$. Moreover, $f(C)$ should be computable in polynomial time, which among other things means we cannot blow up the size of C by more than a polynomial factor.

First of all, let's assume our input is given as a list of gates, where for each gate g_i we are told what its inputs are connected to. For example, such a list might look like: $g_1 = \mathsf{NAND}(x_1, x_3)$; $g_2 == \mathsf{NAND}(g_1, x_4)$; $g_3 = \mathsf{NAND}(x_1, 1)$; $g_4 = \mathsf{NAND}(g_1, g_2)$; .... In addition we are told which gate g_m is the output of the circuit.

We will now compile this into an instance of 3-SAT as follows. We will make one variable for each input x_i of the circuit, and one for every gate g_i. We now write each NAND as a conjunction of 4 clauses. In particular, we just replace each statement of the form "$y_3 = \mathsf{NAND}(y_1, y_2)$" with:

$$\begin{array}{lll}
 & (y_1 \text{ OR } y_2 \text{ OR } y_3) & \leftarrow \text{if } y_1 = 0 \text{ and } y_2 = 0 \text{ then we must have } y_3 = 1 \\
\text{AND} & (y_1 \text{ OR } \overline{y}_2 \text{ OR } y_3) & \leftarrow \text{if } y_1 = 0 \text{ and } y_2 = 1 \text{ then we must have } y_3 = 1 \\
\text{AND} & (\overline{y}_1 \text{ OR } y_2 \text{ OR } y_3) & \leftarrow \text{if } y_1 = 1 \text{ and } y_2 = 0 \text{ then we must have } y_3 = 1 \\
\text{AND} & (\overline{y}_1 \text{ OR } \overline{y}_2 \text{ OR } \overline{y}_3). & \leftarrow \text{if } y_1 = 1 \text{ and } y_2 = 1 \text{ we must have } y_3 = 0
\end{array}$$

Finally, we add the clause (g_m), requiring the circuit to output 1. In other words, we are asking: is there an input to the circuit *and* a setting of all the gates such that the output of the circuit is equal to 1, *and* each gate is doing what it's supposed to? So, the 3-CNF formula produced is satisfiable if and only if the circuit has a setting of inputs that causes it to output 1. The size of the formula is linear in the size of the circuit. Moreover, the construction can be done in polynomial (actually, linear) time. So, if we had a polynomial-time algorithm to solve 3-SAT, then we could solve circuit-SAT in polynomial time too. ■

Important note: Now that we know 3-SAT is **NP**-complete, in order to prove some other **NP** problem Q is **NP**-complete, we just need to reduce 3-SAT to Q; i.e., to show that 3-SAT $\leq_p Q$. In particular, we want to construct a (polynomial-time computable) function f that converts instances of 3-SAT to instances of Q that preserves the YES/NO answer. This means that if we could solve Q efficiently then we could solve 3-SAT efficiently. *Make sure you understand this reasoning — a lot of people make the mistake of doing the reduction the other way around.* Doing the reduction the wrong way is just as much work but does not prove the result you want to prove!

20.3 CLIQUE

We will now use the fact that 3-SAT is **NP**-complete to prove that a natural graph problem called the MAX-CLIQUE problem is **NP**-complete.

Definition 20.2 MAX-CLIQUE*: Given a graph G, find the largest clique (set of nodes such that all pairs in the set are neighbors). Decision problem: "Given G and integer k, does G contain a clique of size $\geq k$?"*

Note that MAX-CLIQUE is clearly in **NP**.

Theorem 20.2 MAX-CLIQUE *is* **NP***-Complete.*

Proof: We will reduce 3-SAT to MAX-CLIQUE. Specifically, given a 3-CNF formula F of m clauses over n variables, we construct a graph as follows. First, for each clause c of F we create one node for every assignment to variables in c that satisfies c. E.g., say we have:

$$F = (x_1 \vee x_2 \vee \overline{x}_4) \wedge (\overline{x}_3 \vee x_4) \wedge (\overline{x}_2 \vee \overline{x}_3) \wedge \ldots$$

Then in this case we would create nodes like this:

$$\begin{array}{llll}
(x_1=0,x_2=0,x_4=0) & (x_3=0,x_4=0) & (x_2=0,x_3=0) & \ldots \\
(x_1=0,x_2=1,x_4=0) & (x_3=0,x_4=1) & (x_2=0,x_3=1) & \\
(x_1=0,x_2=1,x_4=1) & (x_3=1,x_4=1) & (x_2=1,x_3=0) & \\
(x_1=1,x_2=0,x_4=0) & & & \\
(x_1=1,x_2=0,x_4=1) & & & \\
(x_1=1,x_2=1,x_4=0) & & & \\
(x_1=1,x_2=1,x_4=1) & & &
\end{array}$$

We then put an edge between two nodes if the partial assignments are consistent. Notice that the maximum possible clique size is m because there are no edges between any two nodes that correspond to the same clause c. Moreover, if the 3-SAT problem *does* have a satisfying assignment, then in fact there *is* an m-clique (just pick some satisfying assignment and take the m nodes consistent with that assignment). So, to prove that this reduction (with $k = m$) is correct we need to show that if there *isn't* a satisfying assignment to F then the maximum clique in the graph has size $< m$. We can argue this by looking at the contrapositive. Specifically, if the graph has an m-clique, then this clique must contain one node per clause c. So, just read off the assignment given in the nodes of the clique: this by construction will satisfy all the clauses. So, we have shown this graph has a clique of size m iff F was satisfiable. Also, our reduction is polynomial time since the graph produced has total size at most quadratic in the size of the formula F ($O(m)$ nodes, $O(m^2)$ edges). Therefore Max-Clique is **NP**-complete. ■

20.4 Independent Set and Vertex Cover

An Independent Set in a graph is a set of nodes no two of which have an edge. E.g., in a 7-cycle, the largest independent set has size 3, and in the graph coloring problem, the set of nodes colored red is an independent set. The Independent Set problem is: given a graph G and an integer k, does G have an independent set of size $\geq k$?

Theorem 20.3 Independent Set *is* **NP**-*complete.*

Proof: We reduce from Max-Clique. Given an instance (G, k) of the Max-Clique problem, we output the instance (H, k) of the Independent Set problem where H is the complement of G. That is, H has edge (u, v) iff G does *not* have edge (u, v). Then H has an independent set of size k iff G has a k-clique. ■

A *vertex cover* in a graph is a set of nodes such that every edge is incident to at least one of them. For instance, if the graph represents rooms and corridors in a museum, then a vertex cover is a set of rooms we can put security guards in such that every corridor is observed by at least one guard. In this case we want the smallest cover possible. The Vertex Cover problem is: given a graph G and an integer k, does G have a vertex cover of size $\leq k$?

Theorem 20.4 Vertex Cover *is* **NP**-*complete.*

Proof: If C is a vertex cover in a graph G with vertex set V, then $V - C$ is an independent set. Also if S is an independent set, then $V - S$ is a vertex cover. So, the reduction from Independent Set to Vertex Cover is very simple: given an instance (G, k) for Independent Set, produce the instance $(G, n - k)$ for Vertex Cover, where $n = |V|$. In other words, to solve the question "is there an independent set of size at least k" just solve the question "is there a vertex cover of size $\leq n - k$?" So, Vertex Cover is **NP**-Complete too. ■

20.5 Beyond NP

As mentioned earlier, it is an open problem whether $\mathbf{P} \neq \mathbf{NP}$ (though everyone believes they are different). It is also an open problem whether $\mathbf{NP} \neq \textbf{co-NP}$ (though again, everyone believes they are different). One can also define even more expressive classes. For instance, **PSPACE** is the class of all problems solvable by an algorithm that uses a polynomial amount of memory. Any problem in **NP** is also in **PSPACE**, because one way to solve the problem is to take the given instance I and then simply run the verifier $V(I, X)$ on all possible proof strings X, halting with YES if any of the runs outputs YES, and halting with NO otherwise. (Remember, we have a polynomial upper bound on $|X|$, so this uses only a polynomial amount of space.) Similarly, any problem in **co-NP** is also in **PSPACE**. Unfortunately, it is not even known for certain that $\mathbf{P} \neq \mathbf{PSPACE}$ (though all the classes mentioned above are believed to be different). One can also define classes that are *provably* larger than **P** and **NP**. For instance, **EXPtime** is the class of all problems solvable in time $O(2^{n^c})$ for some constant c. This class is known to *strictly* contain **P**. The class **NEXPtime** is the class of all problems that have a *verifier* which runs in time $O(2^{n^c})$ for some constant c. This class is known to strictly contain **NP**. The class of all Turing-computable problems is known to strictly contain all of the above, and some problems such as the Halting problem (given a program $\mathcal{A}$ and an input x, determine whether or not $\mathcal{A}(x)$ halts) are not even contained in that!

20.6 NP-Completeness summary

NP-complete problems have the dual property that they belong to **NP** and they capture the essence of the entire class in that a polynomial-time algorithm to solve one of them would let you solve anything in **NP**.

We proved that 3-SAT is **NP**-complete by reduction from Circuit-SAT. Given a circuit C, we showed how to compile it into a 3-CNF formula by using extra variables for each gate, such that the formula produced is satisfiable if and only if there exists x such that $C(x) = 1$. This means that a polynomial-time algorithm for 3-SAT could solve any problem in **NP** in polynomial-time, even factoring. Moreover, 3-SAT is a simple-looking enough problem that we can use it to show that many other problems are **NP**-complete as well, including Max-Clique, Independent Set, and Vertex Cover.