15-451 Algorithms, Spring 2007

Lectures 11-18

Author: Avrim Blum
Instructor: Manuel Blum

February 20, 2007

Dynamic Programming

Dynamic Programming: Powerful technique that often allows you to solve a problem in time O(n^2) or O(n^3) where a naive approach would take exponential time. Usually, to get speed down below that (if possible), need to add other ideas. Today: 2 or 3 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 bigger ones. Unlike "divide and conquer" (like mergesort or quicksort) it is OK if our subproblems overlap, so long as there aren't too many of them.

Example #1: Longest Common Subsequence

2 strings: S of length n, T of length m. E.g.,
S = ABAZDC
T = BACBAD
LCS is longest sequence of characters that appear left-to-right (but not necessarily in a contiguous block) in both strings.
E.g., here, LCS has length 4: ABAD.

For instance, use in genetics: given two DNA fragments, the LCS gives info about what they have in common and the best way to line them up.

Let's solve the LCS problem using Dynamic Programming. Subproblems: look at LCS of prefix of S and prefix of T. For simplicity, we'll worry first about finding *length* of longest and then modify to produce the sequence.

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

Case 1: what if S[i]!=T[j]? Then, the answer LCS has to ignore one of S[i] or T[j] so LCS[i,j] = max(LCS[i-1,j], LCS[i,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. If I gave you a CS that matched S[i] to an earlier location in T, for instance, you could always match it to T[j] instead. So, the solution is 1 + LCS[i-1,j-1].

So, we can just do two loops, filling in the LCS using these rules. Here's what it looks like pictorially:

	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
```

Just fill out row by row, doing constant amount of work per entry, so O(mn) time overall.

To find sequence: walk backwards through matrix to largest of left or up. If both are < current, then go diagonally and output character. (Or, could modify alg to fill matrix with sequences instead of numbers)

We've been looking at "bottom-up Dynamic Programming". Here's another way of thinking about DP, 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 $T(n)=2T(n-1)+n$. But, suppose that many of the subproblems you reach as you go down the recursion tree are the *same*. I.e., the "tree" is really a "DAG". Then you can hope to get a big savings if you store your computations so that you only compute each one once. Can store these in an array or hash table. Called "memoizing".

E.g., for LCS, 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;
}
```

Memoized version: start by initializing arr[i][j]=unknown for all i,j.

```
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
  return result;
}
```

Running time is O(nm). Why?
 -> we get to the second-to-last line at most n*m times
 => at most 2nm recursive calls.
 => O(nm) running time.

Comparing bottom-up and top-down: top-down (memoized) pays a penalty in recursion overhead, but can be faster for problems where a reasonable fraction of the subproblems never get examined at all, so we can avoid computing them.

Discussion and Extensions
=========================
Equivalent problem: "minimum edit distance", where the legal operations are insert and delete. (E.g., unix "diff", 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| - LCS(S,T) deletes and |T| - 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 DP solution.

==========

Example #2: knapsack problem

Imagine you have a homework assignment with different parts A,B,C,D,E,F,G. Each part has a "value" (in points) and a "size" (time in hours) to do.

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

You have: 15 hours. Which parts should you do? What's the best total value possible? (Assume no partial credit on parts) (A: 34 -- ABFG)

This is called the "knapsack problem":
Given n items. Each has size s_i and value v_i. Knapsack of size S.
Goal: find subset of items of max possible total value such that sum of sizes is <= S.

Can solve in time O(2^n) by trying all possible subsets. Want something better. We'll use DP to solve in time O(n*S).

Let's do this top down by starting with a simple recursive solution and trying to memoize. Start by just computing the best possible value - then we 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 (S <= 0 || 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, time is O(2^n). But, how can we speed up?

Use a 2-d array to store results so we don't end up computing the same thing over and over again. Initialize arr[i][j] to "unknown"

```
      Value(n,S)
      {
        if (S <= 0 || 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;
        }
```

Running time:
 same analysis as for LCS: O(nS).

How to get items? Work backwards: if arr[n][S] = arr[n-1][S] then we *didn't* use the nth item so recursively work backwards from arr[n-1][S]. Otherwise, we *did*, so output nth item and recursively work backwards from arr[n-1][S-s_n].

Can also do bottom up.

==========

Example #3: Matrix product parenthesization

Say we want to multiply 3 matrices X, Y, and Z. And we're going to use the usual algorithm, not Strassen. We could do it like this: (XY)Z or like this X(YZ). Which way doesn't affect final outcome but *does* affect running time to compute it.

E.g., say X is 100x20, Y is 20x100, and Z is 100x20. So, end result will be a 100x20 matrix. If we multiply using usual algorithm, then to multiply lxm matrix by an mxn matrix takes time O(lmn). So in this case, which is better, doing (XY)Z or X(YZ)?

Answer: X(YZ) is better because computing (YZ) takes 20x100x20 steps, and produces a 20x20 matrix, then multiplying this by X takes another 20x100x20 steps, so total is just 2x20x100x20. But, doing the other way, (XY) takes 100x20x100 steps, so already right there you've spent more, and then multplying this with Z takes another 100x20x100 steps.

Question: suppose we need to multiply a series of matrices:
A_1 x A_2 x A_3 x ... x A_n.
What is the best way to parenthesize them?

There are an exponential number of different possible parenthesizations: {2(n-1) choose n-1}/n. So don't want to search through all of them. DP gives us a better way.

Idea: How might you do this recursively? One way: for each possible middle for the final multiplication, optimally solve for the parenthesization of the left and right sides, and calculate the total cost. Then take the best middle. If you keep going, what do the subproblems look like?
 -> For each i,j, what is the best way to multiply A_i x ... x A_j.
In particular,
 -> To figure out how to multiply A_i x ... x A_j, consider all possible middle points k.
 -> For each such k, our cost if we decided to split there would be:
cost-to-multiply(A_i x ... x A_k) <- already computed

```
       + cost-to-multiply(A_{k+1} x ... x A_j) <- already computed
       + cost to multiply the results.   <- get right from the dimensions.
 -> So, just chose the k that minimizes this sum.

So, we store an nxn matrix.  First solve for all with j-i = 1, then
solve for all with j-i = 2, and so on.  At most O(n) choices of k to
consider when filling in any given entry in matrix.

What is the time?
 -> O(n^2) subproblems.
 -> O(n) time per subproblem (because you need to consider O(n) choices of k).
 => O(n^3) time total.
================================================================================
High-level discussion of Dynamic Programming

What kinds of problems can be solved using DP?  One property these
problems have is that if the optimal solution to the problem includes a
solution to a subproblem, then it includes 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 A to C.  Or, in the
knapsack example, if the optimal solution doesn't use item n, then it
is the optimal solution for the problem in which item n doesn't
exist.    The book calls this the "optimal substructure" property.

Sometimes you need to do a little work on the problem to get this
property.  For instance, suppose we're trying to find paths between
locations in Pittsburgh, and some intersections have no-left-turn
rules (this is even worse 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 to have one node per (intersection,direction) pair.
That way you recover the optimal substructure property.
```

15-451 Algorithms 02/22/07 Lecture 12

Graph algorithms I
* Basic notation/terminology
* DFS for Topological sorting
* Dynamic-Programming algs for shortest paths
  - Bellman-Ford, Floyd-Warshall (all-pairs SP)

==========

Graphs
======

A lot of algorithmic problems can be modeled as problems on graphs. Today we will talk about a couple basic ones and we will continue talking about graph algorithms for a lot of the rest of the course.

Reminder of basic terminology:
A graph is a set of NODES, or VERTICES, with edges between some of the nodes. V = set of vertices, E = set of edges. If there's an edge between two vertices, we call them NEIGHBORS. Degree of a vertex is the number of neighbors it has.

standard notation: n = |V|, m = |E|.

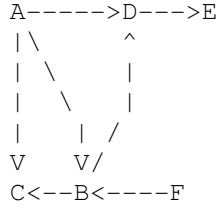
If we don't allow self-loops, what is max number of total edges? {n \choose 2}

In a directed graph, each edge has a direction. Can have up to n(n-1) edges now. For each node, can talk about out-neighbors (and out-degree) and in-neighbors (and in-degree).

==========

TOPOLOGICAL SORTING.

A "DAG" or directed-acyclic-graph is a directed graph without any cycles. E.g.,



Given a DAG, the "topological sorting" problem is: find an ordering of the vertices such that all edges go forward in the ordering. Typically comes up when given a set of tasks to do with precedence constraints (need to do A and F before you can do B), and want to find a legal ordering of performing the jobs.

Here is a nice linear-time algorithm based on depth-first search (can also use it to tell if a directed graph has any cycles).

To be specific, by "DFS of G" we mean "pick a node and do DFS from there. If the whole graph hasn't been visited, then pick an unvisited node and repeat the process. Continue until everything is visited". I.e.,

```
DFS_main(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).
  // exiting node v.
```

How to solve Topological Sorting #2:
1. Do depth-first search of G, and output the nodes as you *exit* them.
2. reverse the order.

Claim: If there is an edge from u to v, then v is exited first. (This implies that when we reverse the order, we have a topological sorting.)

Proof of claim: [Think of u=B, v=D above.] Easy to see if our DFS started at u before starting at v. What if we started at v first? In this case, we would finish v before even starting u since there can't be a path from v to u (else wouldn't be acyclic).

==========

SHORTEST PATHS

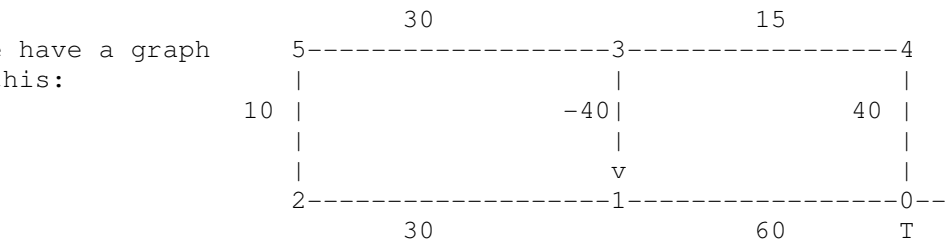
Now going to turn to another basic graph problem, of finding shortest paths, and look at some algs based on Dynamic Programming.

Given: a weighted, directed graph (each edge has a "weight" or "length"), a start node S and a destination node T.
Goal: find shortest path from S to T.

Will allow for negative-weight edges (we'll see later some problems where this comes up when using shortest-path algs as a subroutine) but will assume no negative-weight cycles (else the shortest path has length negative-infinity).

The BELLMAN-FORD Algorithm
==========

(Note: Bellman is the person credited for inventing Dynamic Programming). Algorithm will in fact find the shortest path from all nodes v in the graph to T (not just from S).



How can we use DP to solve? First of all, as usual for DP, let's just compute the *lengths* of the shortest paths first, and then we can reconstruct the paths.

Alg idea:
- find length of shortest path that uses 1 or fewer edges (or infty if none)
  [easy: if v=T, get 0; if v has edge to T, get length of edge; else infty]

- Now, suppose for all v we have solved for length of shortest path to T that

uses i-1 or fewer edges. How can we use to solve for i?

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

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

BELLMAN-FORD pseudocode:
- d[v][i] will be the length of the shortest path from v that uses i or fewer edges (if it exists) or else infinity otherwise. ("d" for "distance")
- we'll have base case be paths of length 0 instead of length 1.

```
initialize d[v][0] = infinity for v != T.  d[T][i]=0 for all i.
for i=1 to n-1:
  for each v not equal to T:
    d[v][i] = MIN ( length(vx edge) + d[x][i-1]  )
              v->x
```

(try it on the above graph!)

Inner MIN operation takes time proportional to out-degree of v. So, inner for-loop takes time O(sum of out-degrees of all nodes) = O(m).
--> total time is O(mn).

How to recosntruct the paths? Work backwards: if you're at vertex v at distance d[v], move to node x such that d[v] = d[x] + len(vx).

==========

Here is a generalization we can also solve using Dynamic Programming. We'll see how to do it in 2 different ways.

ALL-PAIRS SHORTEST PATHS

Say we want to compute length of shortest path between *every* pair of vertices. If we use Bellman-Ford for all n possible starts S, this will take time O(mn^2). Here is a nice algorithm that uses the matrix representation of a graph, and runs in time O(n^3 log n).

ALL-PAIRS SHORTEST PATHS via MATRIX PRODUCTS

Given graph G, define matrix A(G) as follows:
- A[i,i] = 0 for all i.
- if there is an edge from i to j, then A[i,j] = length of that edge.
- otherwise, A[i,j] = infinity.

I.e., A[i,j] = length of 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] = 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*A except we change "*" to "+" and we change "+" to "min".

In other words, instead of computing the sum of products, we compute the min of sums. What if we now want to get shortest paths using 4 or fewer edges? Just need to compute C = B*B. I.e., to get from i to j using 4 or fewer edges, we need to go from i to some k using 2 or fewer edges, and then from k to j using 2 or fewer edges.

Just need to keep squaring O(log n) times.

Running time: O(n^3 log(n)). Need to do log(n) matrix multiplies to get all paths of length n or less and standard matrix multiply takes n^3 operations.

==========

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 the shortest path that's allowed to use node 1 as an intermediate node. Then 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] = len of 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 k or you don't. Total time is O(n^3).

What's amazing here is how compact and simple the code is!

==========

15-451 Algorithms 02/27/07 Lecture 13

Graph Algorithms II
* Dijkstra's alg for shortest paths
* Maximum bottleneck path
* Minimum Spanning Trees: Prim and Kruskal

Midterm in 1 week
1 sheet of notes, no book

==========

Shortest paths
==========

Given graph G, start node S, destination T, want to find shortest path from S to T.

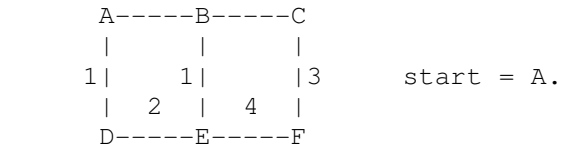
In the last class, we saw a couple Dynamic-Programming algorithms for shortest path problems. In particular, the Bellman-Ford algorithm solved for the shortest path tree from a start node S in time O(mn).

Def: the SHORTEST PATH TREE from a start node S is a tree giving the shortest path from S to every other node in G.

Q: assuming all other nodes are reachable from S, why must such a tree exist? That is, why must we be able to describe the shortest paths from S to every other node using just a single tree? A: because if shortest path from S to u goes through v, then it uses a shortest path from S to v. So, for each node u, just need to indicate the very last edge on a shortest path to u. [This is also an "optimal subproblem" property that allows for dynamic programming to work.

First alg for today (which you've seen in 15-211) is an algorithm for building a shortest path tree that's faster than Bellman-Ford, called Dijkstra's algorithm. However, it requires that all edge lengths be non-negative. See if you can figure out where proof of correctness of alg requires non-negativity.

Example for illustration:



DIJKSTRA's ALG
---

initialize:
- tree = {start}. No edges. label start as having distance 0.

invariant: nodes in tree have correct distance to start.

repeat:
- for each nbr x of tree compute an (over)-estimate of distance to start:

$$\text{distance}(x) = \min_{\substack{\text{edges } e=(v,x) \\ v \text{ in tree}}} [\ \text{distance}(v) + \text{length}(e)\ ]$$

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.

- put x of minimum distance into tree, connecting it via the argmin
      (the edge e used to get distance(x))

[do above example]

Claim: even if some of distances are wrong, the *minimum* one is correct.

Proof: say x is neighbor of tree of smallest distance(x). What does the *true* shortest path to x look like? The key point is that the last edge this path takes must come directly from the tree. Let's argue this by contradiction. Suppose instead the first non-tree vertex on the shortest path to x is some node y!=x, and then the path goes from y to x. Then the length of the path must be at least distance(y), and by definition, distance(x) is smaller (or at least as small if there are ties). [This is where the "non-negativity" comes in. Technically, the wording of our proof was a little sloppy if we allow zero-length edges: think how you would re-word it for that case.]

Running time: if you use a heap and code this up right, you can get time of O(m log n). Start with all nodes on heap, everyone having distance of infinity except start having distance of 0. Repeatedly pull off the minimum, then update its neighbors, tentatively assigning parents any time the distance of some node is lowered. Takes linear time to initialize, but then we do m updates, at O(log n) cost each. If you use something called a "fibonacci heap" (that we're not going to talk about) can get time down to O(m + n log n)

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 to solve this??

Def: width of path = minimum width of edges on the path
bottleneck(v) = width of widest path to v.

Now, we'd update:

$$\text{bottleneck}(x) = \max_{\substack{\text{edges } e=(v,x) \\ v \text{ in tree}}} [\min(\text{bottleneck}(v), \text{width}(e))]$$

- put x of maximum bottleneck into tree, connecting it via the argmax

We'll actually use this later in the course...

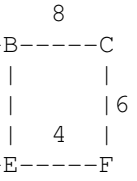
==========

MINIMUM SPANNING TREE

A SPANNING TREE is a tree that touches all the nodes of a graph. (So, it only makes sense in a connected graph.) A Minimum Spanning Tree is the shortest spanning tree (size of tree is sum of its edge lengths)

For instance, can imagine you're setting up a communication network.

For example:



What is the MST here?

(Note: our defn is only for undirected graphs).

Prim's algorithm

Prim's algorithm is the following simple MST algorithm:
- pick some arbitrary start node.
- add shortest edge to current tree.
- continue until tree spans all nodes.
--> what does this do on above graph?

[So, a lot like Dijsktra, except just put in the shortest edge rather than adding the length to the distance of endpoint to the start node. But even though the alg is simpler than Dijkstra, the analysis is a bit trickier...]

Now we need to prove that this works. Will prove by induction.

First: useful fact about spanning trees: If you take any spanning tree and add an edge to it, you get a cycle. That's because there was already one path between the endpoints, and now there are two. If you then remove any edge in the cycle, you get back a spanning tree.

Proof of correctness of Prim's alg:

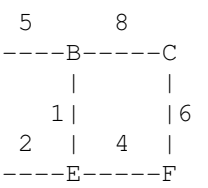
> Proof by contradiction. Say it fails. Consider the first edge "e" chosen by the algorithm that is not consistent with an MST. Let T be the tree we have built just before adding in e, and let M be the MST consistent with T.
>
> Now, suppose we add e to the M. This creates a cycle. Since e has one endpoint 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 length(e') >= length(e) by definition of the algorithm. So, if we add e to M and remove e', we get a new tree that is no larger than M was, contradicting our defn of "e".

Running time: To make this efficient, we need a good way of storing the edges into our current tree so we can quickly pull off the shortest one. If we use a heap, we can do this in log(n) time. So, the total time is O(m log n) where m is the number of edges and n is the number of vertices.

Kruskal's algorithm
==========
Here's another nice algorithm, called Kruskal's algorithm.

Algorithm: Sort edges by length and examine them from shortest to longest. Put edge into current forest (forest is set of trees) if it doesn't form a cycle.

E.g., look at in this graph:



Proof of correctness: Can basically use same argument as before. Let e be the first edge we add that is inconsistent with any MST, and let F be the forest just before adding e. Let M be an MST consistent with F. Put e into M. This makes a cycle. Go around until you take some edge e' jumping between trees in F. By definition of our algorithm, len(e) <= len(e'). So, putting in e and removing e' can only make M better, which is a contradiction to our defn of e.

What about running time? Takes O(m log m) to sort. 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. Turns out there's a nice data structure called "union-find" data structure for doing this. So fast that it actually will be low-order compared to the sorting.

Simple version of Union-Find: total time O(m + n log n) for our series of operations. More sophisticated version of Union-Find: total time O(m lg^*(n)) -- or even O(m alpha(n)), where alpha(n) is the inverse-Ackermann function that grows even more slowly than lg^*.

-> lg^*(n) is the number of times you need to take the lg until you get down to 1. So,

```
lg^*(2) = 1
lg^*(2^2 = 4) = 2
lg^*(2^2^2 = 16) = 3
lg^*(2^2^2^2 = 65536) = 4
lg^*(2^65536) = 5
```

[I won't define inverse-Ackerman, but to get alpha(n) to 5, you need n to be at least a stack of 256 2's.]

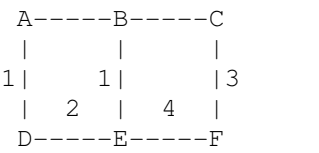
* Maximum bottleneck path
* Minimum Spanning Tree recap: Prim and Kruskal Midterm on Tues
* Midterm review 1 sheet of notes, no book

==========

One nice thing about Dijkstra's algorithm is that you can modify it to solve other related problems. (This kind of thing often makes a good test question)

DIJKSTRA's ALG recap (finds shortest path tree)

E.g.,



Initialize:
- tree = {start}. No edges. label start as having distance 0.

Invariant: nodes in tree have correct distance to start.

Repeat:
- for each nbr x of tree compute an (over)-estimate:

$$dist(x) = \min_{\substack{\text{edges } e=(v,x) \\ v \text{ in tree}}} [\ dist(v) + length(e)\ ]$$

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.

- put x of minimum distance into tree, connecting it via the argmin (the edge e used to get dist(x))

Actual implementation: just update dists of neighbors of the node put onto tree, since they are the only ones that can change.

Maximum-bottleneck path

Now consider the following problem called the "maximum bottleneck path" problem. Imagine the edge weights represent capacities of the edges (like "widths" rather than "lengths") and you want the path between two nodes whose minimum capacity is largest. How can we solve this?

Def: capacity of path = minimum capacity of edges on the path
cap(v) = capacity of max capacity path to v.

Now, we'd update:

$$cap(x) = \max_{\substack{\text{edges } e=(v,x) \\ v \text{ in tree}}} [\min(cap(v), w(e))]$$

- put x of maximum cap(x) into tree, connecting it via the argmax

We'll actually use this later in the course...
==========

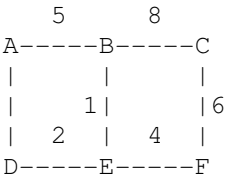
MINIMUM SPANNING TREE

A SPANNING TREE is a tree that touches all the nodes of a graph. (So, it only makes sense in a connected graph.) A Minimum Spanning Tree is the shortest spanning tree (size of tree is sum of its edge lengths)

Prim's algorithm
==========

Prim's algorithm is the following simple MST algorithm:
- pick some arbitrary start node.
- add shortest edge to current tree.
- continue until tree spans all nodes.

For example:



[So, a lot like Dijsktra, except just put in the shortest edge rather than adding the length to the distance of endpoint to the start node. But even though the alg is simpler than Dijkstra, the analysis is a bit trickier...]

Now we need to prove that this works. Will prove by induction.

> Assume inductively that there exists an MST M that contains our tree T built so far. Want to show there exists an MST that contains T *and* the new edge e we are about to add. If M contains e then we're done. If not, then add e to M. 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 length(e') >= length(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 both T and e.

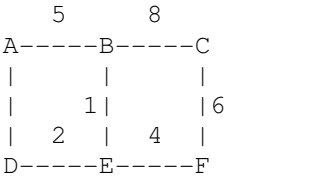
Running time: To make this efficient, we need a good way of storing the edges into our current tree so we can quickly pull off the shortest one. If we use a heap, we can do this in log(n) time. So, the total time is O(m log n) where m is the number of edges and n is the number of vertices.

Kruskal's algorithm
==========

Here's another nice algorithm, called Kruskal's algorithm.

Algorithm: Sort edges by length and examine them from shortest to longest. Put edge into current forest (forest is set of trees) if it doesn't form a cycle.

E.g., look at in this graph:



Proof of correctness: Can basically use same argument as before. Assume by induction there exists a MST M containing all edges in our forest F built so far. Want to show there exists an MST M' containing F and the next edge e we are about to include. If M contains e then we are done. Else put e into M. This makes a cycle. Go around until you take some edge e' jumping between trees in F. By definition of our algorithm, len(e) <= len(e'). So, putting in e and removing e' to get M' can only make M better.

What about running time? Takes O(m log m) to sort. 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. Turns out there's a nice data structure called "union-find" data structure for doing this. So fast that it actually will be low-order compared to the sorting.

Simple version of Union-Find: total time O(m + n log n) for our series of operations. More sophisticated version of Union-Find: total time O(m lg^*(n)) -- or even O(m alpha(n)), where alpha(n) is the inverse-Ackermann function that grows even more slowly than lg^*.

-> lg^*(n) is the number of times you need to take the lg until you get down to 1. So,

```
lg^*(2) = 1
lg^*(2^2 = 4) = 2
lg^*(2^2^2 = 16) = 3
lg^*(2^2^2^2 = 65536) = 4
lg^*(2^65536) = 5
```

[I won't define inverse-Ackerman, but to get alpha(n) to 5, you need n to be at least a stack of 256 2's.]

Simple UNION-FIND O(m + n log n)-time.
==========

General picture is we maintain a collection {S_1, S_2, ..., S_k} of components, initially each with one element. Operations:

```
Union(x,y) - replace the component S_x containing x and the S_y
             containing y with the single component S_x U S_y.
Find(x) - return the unique ID for the component containing x (this
          is just some representative element in it).
```

Given this, when we consider some edge (v,w) we just see if Find(v)==Find(w). To put in the edge(v,w) we do a union.

Algorithm

Each set will be just a linked list: each element has pointer to next element in list. But, augment so that each element also has a pointer directly to head of list. Head of list is the representative element.

Initialization: for each x, let x->head = x. Total time O(n).

Find(x) - constant time: return(x->head). We do this O(m) times, so total time O(m).

Union(x,y) - to union the two lists (let's call them A and B with lengths L_A and L_B). We append B onto end of A. What is is the time needed? Answer: we first have to walk down A to get to bottom element. Then hook up head of B to this. Then walk down B updating the pointers to the head. So total time is O(L_A + L_B)

Q: Can we get just O(L_B)?

Method 1: splice B into the middle of A, at x.
Method 2: if store pointer to tail in the head, this also avoids walking down A.

Q: Can we reduce this to O(min(L_A,L_B))?

A: Yes. Just store length of list in the head. Then compare and append shorter one onto end of longer one. Then update the length count.

Claim: this gives us the O(m + n log n) total running time.

PROOF: We already analyzed the initialization and find operations. Each union costs O(length of list we walk down). Key idea: let's pay for this by charging O(1) to each element that we walked over. Think of as a charge for updating its head pointer.

Now, let's look at this from the point of view of some lowly element x sitting in a list somewhere. Over time, how many times does x get stepped on and have its head pointer updated? Answer: at most O(log n) times. Reason: because every time x gets stepped on, the size of the list he is in doubles (at least).

So, we were able to pay for unions by charging elements stepped on, and no element gets charged more than O(log n), so total cost is O(n log n) for unions, or O(m + n log n) overall.

==========

MIDTERM REVIEW
==========

- Last year's midterm now up on webpage.

- Main topics:

 - recurrences
 - probabilistic reasoning
 - reasoning about upper and lower bounds

- binary search trees
- hashing
- Dynamic programming
- MST and shortest paths

* Graphs Algorithms III: fun with BFS and DFS
 - 2-coloring
 - properties of BFS and DFS trees
 - strongly-connected components

==========

Problem 1: Graph Coloring

Given a graph, want to give each node a color such that no two neighbors have the same color. Goal is to do this with the fewest total number of different colors (easy to do with n colors).

E.g., pentagon.

E.g., consider the problem of scheduling final exams into the fewest number of different time-slots. Each node is a class, and there is an edge between two if lots of students are taking both classes. In compilers, this kind of problem comes up when assigning variables to registers.

Unfortunately, graph-coloring problem is NP-hard. In fact, even telling if possible to color with <= 3 colors is NP-hard in general. But, 2-coloring problem (color with 2 colors if possible else say "impossible") is easy. How to do it?

Turns out it's pretty simple. Pick some vertex v and give it one color (say, red). Then all its neighbors had better be the other color (say, blue). All their neighbors had better be red, their neighbors blue and so forth, sweeping across the graph until everything in v's connected component has a color. Now, we check: if there is any bad edge (both endpoints the same color) then G is not 2-colorable: why? because all our decisions were forced. Otherwise we have a legal 2-coloring of v's component and we run this procedure again on any remaining components (which don't interfere with each other since there are no edges between components).

Another term for a graph being 2-colorable is that it is bipartite. So, this is an algorithm for testing if a graph is bipartite, and if so, finding a bipartition.

One way to think of this is we are doing breadth-first search. (can actually use DFS to do this too, but BFS is easier to visualize).

Reminder about BFS and discussion of BFS trees and DFS trees

Breadth-first search: start at some vertex v. Then go to all neighbors (call them "level 1" since they're distance 1 from v) then to their unvisited neighbors (level 2 at distance 2 from v) and so on, sweeping this wavefront across the graph. Operationally, we can do this by storing current wavefront in a queue. Something like

```
BFS(s):
   mark s as visited and insert s into queue q.  level(s)=0.
   while (queue not empty):
      v = q.remove()
```

```
      for all unvisited neighbors w of v:
        put edge (v,w) into BFS tree T
      mark w as visited, level(w)=level(v)+1
      q.insert(w)
```

Total time = time for BFS = O(m + n)

Note 1: There are multiple possible BFS trees from a given start s, depending on ordering of neighbors. [example] Same with DFS.

Note 2: What do edges in G that are *not* in the BFS tree look like? If G is undirected, then any such edge (u,v) must be between two vertices at the same or adjacent levels. In particular, no back-edges, only cross-edges. [not true if G is *directed*]

In particular, with respect to coloring, if coloring was illegal there must be an edge in G between two nodes at the same level i.

How about DFS trees? Let's write out DFS again since we're going to use it later anyway. Also introduce notion of "DFS numbering" which is just the order in which vertices are first visited by the DFS.

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

DFS(v):
  mark v as visited. // entering node v
  dfsnum(v) = count++.
  for each unmarked out-neighbor w of v:
      put (v,w) into DFS tree T
      do DFS(w).
  mark v as exited.
```

In an *undirected* graph, what do the non-tree edges of G look like? Can you have cross-edges? [between u and v such that neither is an ancestor of the other] No. Only back-edges.

In a *directed* graph, you *can* have cross-edges, but they have to point to a node of lower dfsnum that was already exited.

Say we are at some node v, what can we say about lower dfsnum nodes that have *not* yet been exited? They are exactly the ancestors.

==========

PROBLEM 2: strong connectivity.

A directed graph G is strongly-connected if you can get from any node to any other node. Equivalently, for every unordered pair {u,v}, you can get from u to v and v to u (they are mutually-reachable).

Linear-time alg to tell if G is strongly-connected?
- pick some node v.
- see if v can reach everybody (DFS or BFS).
- compute reverse graph G^R (reverse all edges)
- do it again (i.e., see if every vertex can *reach* v in G).

If either one fails (v can't reach everybody or not everybody can reach v) then clearly G is not strongly-connected. If both succeed, then G *is* strongly-connected. Why?

==========

PROBLEM 3: strongly-connected components (SCCs).

Given any graph G, can decompose into strongly-connected components.

Def: u and v are in the same SCC if u and v are mutually-reachable.

Claim: notion of SCCs is well-defined: if u and v are in same SCC and v and w are in same SCC then u and w are in same SCC. Why?

[insert example]

So, the SCCs form a partition of the vertices of G. If you view each SCC as a "super-node" then what's left is a DAG. Why?

In fact, if you run depth-first-search on G, it will behave at the high-level like it is running on this DAG. In particular, once you enter some SCC, say at vertex v, you won't exit your recursive call until you have visited all of v's component. This suggests we might be able to use DFS to actually *find* all the SCCs, which in fact we can, though it ends up being pretty tricky.

Let's define the "head" of an SCC to be the first vertex v in it reached by the DFS. Let's now give some observations.

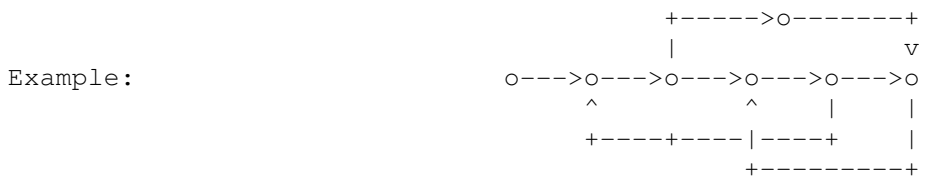
Observation #1: Each SCC is a connected subtree of the DFS tree, with its head as the root of the subtree. [Why can't the DFS go from the current component, off into some other component and then back before popping?] So, if you chopped the tree at every edge leading into a head, you would decompose it into the strongly connected components. So, all we need to do is identify the heads, then we can chop away...

Observation #2: v is head of an SCC iff v cannot reach a lower dfsnum w that can also reach v.

Now, let's try to use these to construct an algorithm. First of all, our goal will be to identify the heads as they are exited in the DFS. When we do identify a head, let's at that point output the SCC and mark all vertices in it as "outputted" (can do in time proportional to the size of the component). So we will be outputting SCCs in reverse topological order. One more observation:

Observation #3: Assuming we are acting as claimed, all vertices visited but not yet outputted are either in the current SCC or in some ancestor SCC. In particular, any such vertex can reach our current node!

So, this would be great if we could somehow get DFS to output the lowest reachable non-outputted vertex when it returns. Could then use to identify v as head when it's time to exit and chop away. Unfortunately, this is not so easy....



But, what if we change "reachable" to "reachable with only at most non-tree edge".

Define low(v) = lowest dfsnum non-outputted vertex reachable from v by a path that uses at most one non-tree edge. [by "non-outputted we mean not yet outputted by the time we exit v]

This we *can* get DFS to return, since we can compute it at the leaves of the DFS tree, and if we know the answers for all children of v, we can compute it for v.

Claim: low numbers are sufficient. v is head iff low(v)=v.

Direction 1: if low(v) < v then v is not head.
Proof: Let w = low(v). w is not yet outputted so by observation 3, it can reach v. So by observation 2, v is not a head.

Direction 2: if v is not head then low(v) != v.
Proof: if v is not head then there must be a path from subtree of v to an ancestor of v. This entire path must be inside v's SCC, so the first back- or cross-edge on it must be to an unoutputted vertex of lower dfsnum.

* network flow
* bipartite matching

==========

Today and next time we are going to talk about an important algorithmic problem called the Network Flow problem. It's important because it can be used to express and solve a lot of different things. In O.R. they have entire courses devoted to the topic.

NETWORK FLOW

- directed graph. Source node. Sink node. Each (directed) edge has a capacity. (Let's say these are integers.) Goal is to flow as much stuff as possible from source to sink [not just one path]. E.g.,

s-->a, cap = 4. a-->c, cap=3. c-->t, cap = 2.
s-->b, cap = 2. b-->d, cap=3. d-->t, cap = 4.
c-->b, cap = 1. b-->c, cap=2.

E.g., want to route message traffic from source to sink, and the capacities tell us how much bandwidth we're allowed on each edge.

Formally, rules are:
Capacity constraint: on any edge, f(u,v) <= c(u,v)
Flow conservation: for any vertex except s and t, flow in = flow out.

What is the max flow here? Answer: 5.

How can you see that above max flow was really maximum? Notice, this flow saturates the a-->c and s-->b edges. And, if you remove these, you disconnect t from s. In other words, the graph has a "s-t cut" of size 5. (a set of edges of total capacity 5 that if you remove disconnects the source from the sink). So, the point is that any unit of stuff 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, max flow can't be larger than any s-t cut.

DEFINITION: an s-t cut is a set of edges whose removal disconnects t from s. Or, formally, a cut is a partition of vertex set into A and B where s is in A and t is in B. (The edges of the cut are then all edges going from A to B).

DEFINITION: the capacity of 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 edges from B to A.)

Easy fact we showed above: max s-t flow <= min s-t cut.

How to find a maximum flow and prove it's 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 graph and repeat until there is no longer any path with capacity left to push any more flow on. Of course, we need to *prove* that this works: we can't somehow end up at a suboptimal solution by making bad choices along the way.

This is called the Ford-Fulkerson algorithm.

Basic Ford-Fulkerson algorithm

While there exists an s-->t path P of positive residual (leftover) capacity:
   push maximum possible flow along P (saturating at least one edge on it)
[These are called "augmenting paths"]

Definition: "RESIDUAL CAPACITY" c_f(u,v) = c(u,v) - f(u,v)
      where we make the convention that f(v,u) = -f(u,v).

Definition: the RESIDUAL GRAPH is the directed graph with all edges of
        positive residual capacity, each one labeled by its residual
        capacity. Note: may include "back-edges" of original graph.

--> Example on above graph: start with path s-->b-->c-->t. Draw flow using
    "[flow/capacity]" notation, and draw residual graph.

Note: residual capacity can be *larger* than original capacity if we
had flow going in opposite direction. E.g., residual capacity on the
c-->b edge is now 3.

CLAIM: Any legal flow in the Residual Graph can be added to the flow
so far and get a legal flow in the original graph.

PROOF: (1) flow-in = flow-out condition is always satisfied when you
add two flows. (2) Capacity constraints satisfied by defn of residual.

Why is the residual graph good to have? One reason: now left with
same type of problem we started with. Can use any black-box
path-finding method (e.g., DFS).

--> Note: running time is not necessarily so great - it depends on the
choices we make. E.g., standard bad-choice graph. After we prove
correctness, we'll worry about about algorithms to fix this problem
(next time).

Algorithm by design finds a legal flow. Why is it maximum?

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
component containing s and B = rest. Now, let C = 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 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 A-->B we added k to flow from A to B (and subtracted k from
the residual capacity) and every time we went B-->A we took away k
from this flow (and added k to the residual capacity). And, we went
from A-->B exactly one more time than we went from B-->A. So, the
residual capacity 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 C must be a minimum cut.

Note, we've actually proven the nonobvious MAXFLOW-MINCUT theorem:

In any graph, max flow from s to t = capacity of minimum (s,t)-cut.

We started with saying max flow <= min cut. But now we've argued that this algorithm finds a flow of value equal to SOME cut, so it can't be less than the MINIMUM.

We have also proven the INTEGRAL-FLOW THEOREM: if all capacities are integers, then there is a max 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 hwk 5.]

Bipartite Maximum matching problem

Say we wanted to be more sophisticated about assigning homework presentation slots to groups. We could ask each group to list the slots acceptable to them, and then draw this as a bipartite graph:

group	slot
A	1
B	2
C	3

Edge if acceptable: e.g, (A 1) (A 2) (B 1) (B 3) (C 2) (C 3)

This is a BIPARTITE GRAPH: a graph with two sides L and R, and all edges go between L and R.

A MATCHING is a set of edges with no endpoints in common. What we want here is a PERFECT MATCHING - a matching that connects every point on the left hand side with some point on the right hand side. What's a perfect matching here? More generally (say there is no perfect matching) we want a MAXIMUM MATCHING - a matching with maximum possible number of edges.

Algorithm to solve:
---

1. Set up fake "start" node S connected to all in L. Connect all in R to a fake "end" 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 matching corresponding to flow.

This finds a legal matching because edges from R to T have capacity 1, so the matching can't have two edges into same node, and similarly the edges from S to L have cap=1, so you can't have two edges leaving same

node in L. It's maximum because any matching gives you a flow of the same value. (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 #edges in the matching since each augmenting path gives us 1 new edge.

[Run alg on above example]. Notice a neat fact: say you start matching A->1, C->3. These are bad choices. But, the augmenting path automatically undoes them as it improves the flow!

Matchings come up as subroutines in a lot of problems like matching up suppliers to customers, or cell-phones to cell-stations when you have overlapping cells. Also as a basic subroutine in other algorithmic problems.

Network Flow II
- Edmonds-Karp #1
- Edmonds-Karp #2
- further improvements...
- min-cost max flow

==========

NETWORK FLOW RECAP: Given a directed graph G, a source s, and a sink t. Each edge (u,v) has some capacity c(u,v). Goal is to find the maximum flow possible from s to t.

Example: 4 nodes {s,a,b,t}. s-->a has cap 4, a-->t has cap 2, s-->b of cap 2, b-->t of cap 3, and a-->b of cap 3. (all non-edges can be viewed as edges of capacity 0).

Last time: looked at Ford-Fulkerson algorithm. Used to prove maxflow-mincut theorem, also integral flow theorem.

Key idea: Given a flow f so far, we describe the capacities left over in a "residual graph". Specifically, the residual graph has all edges of positive "residual capacity", defined as:

$$c_f(u,v) = c(u,v) - f(u,v)$$

where we define

$$f(v,u) = -f(u,v)$$

FF alg is: find path of positive residual capacity; push flow to saturate; repeat.

Point of this is that any legal flow in the residual graph, when added to f (edge by edge) is a legal flow in the original graph (satisfies capacity constraints; obeys flow in = flow out from each node). And, this is true vice-versa. This means that the maximum flow in the residual graph, plus f, is a maximum flow in the original graph.

[run algorithm on example. Draw flow using flow/cap notation in orig graph]

IMPROVING THE RUNNING TIME OF NETWORK FLOW

==========

Assuming capacities are integers, the basic Ford-Fulkerson algorithm could make up to F iterations, where F is the value of the max flow. Each iteration takes O(m) time to find a path (e.g., DFS or BFS) and to compute the residual graph. So, total time is O(mF).

This is fine if F is small, like in the matching case. Not good if capacities are in binary and F could be very large.

[Note: let's assume graph has been pre-processed to delete any disconnected parts. So, m >= n-1, and can say DFS runs in time O(m) instead of O(m+n)].

Edmonds-Karp #1

Idea: don't just choose an arbitrary path. Instead, pick the path of largest (or approximately largest) capacity.

Two versions:
(1.1) use maximum bottleneck path (path with largest residual capacity). We saw how to do this using a Prim-like algorithm in time O(m log n).

(1.2) Define a "capacity threshold" c, and then just try to find an s-t path of residual capacity >= c. Advantage of this: can do in O(m) time. Just do DFS over edges of residual capacity >= c.

How to set c? Can start at max capacity over all edges, and if that fails you then divide by 2. This guarantees you are within a factor of 2 of max capacity path. You waste some time with the unsuccessful runs, but you have at most O(log C) of them in total (C = max edge capacity). Can reduce to O(log F) using doubling idea from midterm.

Claim: Either way, this causes FF to make at most O(m log F) iterations. So total time using version 1.2 is O(m^2 log F).

Proof of claim: First, if the current residual graph has max flow f, then if we drop all edges of res cap < f/m, there must still exist a path from s to t (why? because let A be the component containing s. If doesn't have t, then since there at most m edges out of A, this would be a cut < f). So, max bottleneck path has capacity at least f/m, and (if using version 1.2) this means c is at least f/(2m).

Let's just focus on version 1.2 (since then 1.1 follows directly). We have c is at least f/(2m). So can have at *most* 2m iterations before c gets cut down by factor of 2. Since c <= F at the start, we can cut down c at most log(F) times, so total number of iterations is at most 2m*log(F).

Can we remove dependence on F completely?

EDMONDS-KARP #2

Edmonds-Karp #2: always pick the *shortest* path (rather than the max-capacity path)

Claim: this makes at most mn iterations. So, run time is O(nm^2) since we use BFS in each iteration. Proof is pretty neat.

Proof: Let d(t) be the distance from s to t in the current residual graph. We'll prove the theorem by showing that every m iterations, d(t) has to go up by at least 1.

Let's lay out the graph in levels according to a BFS from s. I.e., nodes at level i are distance i away from s, and t is at level d(t). Now, notice that so long as d(t) doesn't change, the paths found WILL ONLY USE FORWARD EDGES IN THIS LAYOUT. Each iteration saturates (and removes) at least 1 forward edge, and adds only backward edges (so no distance ever drops). This can happen at most m - d(t)+1 <= m times since after that many times, t would become disconnected from s. So, after m iterations, either t is disconnected (and d(t) = infinity) or else we must have used a non-forward edge, implying that d(t) has gone

up by 1.

Since d(t) can increase at most n times, there are at most nm iterations.

DINIC & MPM
==========
Can we do this a little faster? (this part won't be on any tests and it's not crucial you get the details, but the idea is pretty nice.)

Previous alg: mn iterations at O(m) each for O(m^2 n) total. Let's see if we can reduce to O(mn^2) and finally to O(n^3). Idea: Given the current BFS layout (also called the "level graph"), we'll try in O(n^2) time all at once to find the max flow that only uses forward edges. This is sometimes called a "blocking flow". Guarantees that when we take the residual graph, d(t) has gone up by 1. So, <= n iterations.

Define c(v) = capacity of vertex v to be: min[c_in(v), c_out(v)], where c_in(v) is sum of capacities of in-edges and c_out is sum of capacities of out-edges.

The Algorithm:

- In O(m) time, create BFS layout from s, and intersect with backwards BFS from t. Compute capacities of all nodes.

- Find 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.

- if c is not zero, then we greedily pull c units of flow from s to v, and then push it along to t. This seems like just another version of the original problem. But, there are two points here:

  (1) 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 BFS forward from v and backward from v, pushing the flow level-by-level, so we never examine any edge twice. [do example] 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.

  (2) To get O(n^2), 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 through edges that get saturated, and (b) time spent on edges that we didn't quite saturate [at most one of these per node]. We only take time O(n) on type-b operations. Type-a operations result in deleting the edge, so the edge won't be used again in the current level graph. So over *all* vertices, the *total* time of these is just O(m).

So, our running time per iteration is O(n^2 + m) = O(m). [O(n^2) for the type-b and O(m) for the type-a] Once we've saturated the level graph, we recompute it (in O(m) time, but this can be put into the O(n^2)) and re-solve. Total time is O(n^3).

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 $0 to match to there, their second choice is slot 2 so it costs us $1 to match the group here, and it can't make slot 3 so it costs $infinity to match 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.

Generalization to flows is called the MIN-COST MAX FLOW problem.

Here, each edge has a cost w(e) as well as a capacity c(e). Cost of a flow is the sum over all edges of the flow on that edge times the cost of the edge. I.e.,

cost of flow f = SUM_e w(e)*f(e).

Goal is to find, out of all possible maximum flows, the one with the least total cost. Can have negative costs (or benefits) on edges too.

- These are more general than plain max flow so can model more things.

There are a couple ways to solve this. One way is to add an edge from t to s of infinite capacity and large negative cost, and to convert this into the "MIN-COST CIRCULATION" problem. Basically, by doing this, we now can just have the goal of finding a circulation (a flow that obeys flow-in = flow-out at EVERY node (no s and t anymore)) that has smallest cost (this will be a negative number). This will be the min-cost max-flow in the original graph.

Algorithm: find a negative cost cycle and saturate it.

Can use Bellman-Ford to find negative cost cycles.

Running time will be like Ford-Fulkerson. To speed up, ideally would like the "most-negative-cost cycle" but that's NP-hard.
Instead there are various tricks you can use to still do well. E.g., in the Goldberg-Tarjan algorithm, you instead find the cycle that minimizes (cost of cycle)/(# edges in cycle), which is something you *can* solve. [add some value k to all edges and check if there's still a negative-cost cycle using BF. Do binary search to find the value of k s.t. the min-cost cycle has value 0. That's the one that minimizes (cost of cycle)/(# edges in cycle) in the original graph.]

Then you prove this has a bound roughly like Edmonds-Karp #1.

* Linear programming

==========

In last couple of classes 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 do algorithmically: LINEAR PROGRAMMING. (Except we won't necessarily be able to get integer solutions, even when 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. Especially problems of allocating resources and a lot of business applications. In GSIA/Tepper there are entire courses devoted to linear programming. We're only going to have time for 1 lecture. So, 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, motivate with an example:

There are 168 hours in a week. Say we want to allocate our time between classes and studying (S), fun activities, relaxing, going to parties (P), and everything else (E) (eating, sleeping, taking showers, etc). To survive need to spend at least 56 hours on E (8 hrs/day). To maintain sanity need P+E >= 70. To pass courses, need S >= 60, but more if don't sleep enough or spend too much time partying: 2S+E-3P >= 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: 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*.

This is called a *linear program*. What makes it linear is that all our constraints were linear in our variables. E.g., 2S+E-3P>=150. And our objective function is also linear. Not allowed things like requiring S*E >= 100, since this wouldn't be linear.

More formally, here is the definition of the linear programming problem LINEAR PROGRAMMING:

==========

* Given n variables x_1,..., x_n.
* Given m linear inequalities in these variables (equalities OK too):

E.g., $3x_1 + 4x_2 <= 6$, $0 <= x_1 <= 3$, etc.

* May also have an objective function: $2x_1 + 3x_2 + x_3$.

* Goal: find values for the x_i's that satisfy the constraints and maximizes the objective.

"feasibility problem": no objective, just want to satisfy the constraints.

For instance, let's write out our time allocation problem this way.
Variables are S, P, E.
Objective is to maximize 2P+E subject to these constraints:

```
        S + P + E = 168
        E >= 56
        S >= 60
        2S+E-3P >= 150
        P+E >= 70
        P >= 0    (can't spend negative time partying)
```

MORE EXAMPLES OF MODELLING PROBLEMS AS LINEAR PROGRAMS:
==========

Typical Operations-Research kind of problem (from Mike Trick's course notes):
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

We need to produce at least 400 cars at plant 3 according to labor agreement. Have 3300 hours of labor, 4000 units of material available. Allowed to produce 12000 units of pollution. Want to maximize number of cars produced. How to model?

First step: What are the variables?
x_1,x_2,x_3,x_4, where x_i = # cars at plant i.

Second step: What is our objective?
maximize $x_1+x_2+x_3+x_4$

Last step: what are the constraints?

```
x_i >= 0 for all i
x_3 >= 400
2x_1 + 3x_2 + 4x_3 + 5x_4 <= 3300
3x_1 + 4x_2 + 5x_3 + 6x_4 <= 4000
15x_1 + 10x_2 + 9x_3 + 7x_4 <= 12000
```

MAX FLOW
========
Can model this as LP too.

Variables: set up one variable x_e per edge e. Let's just represent

the positive flow.

Objective: maximize $\sum_{\text{edges } e \text{ into } T} x_e$

Constraints:

For all e, $x_e \le c_e$ and $x_e \ge 0$. [c_e is capacity of edge e]

For each node v except S and T,

$$\sum_{\text{edges } e \text{ into } v} x_e = \sum_{\text{edges } e' \text{ leaving } v} x_{e'}$$

Also, delete edges exiting T, or else add the constraint that no flow is allowed to be on them. Otherwise, the solution can fool the objective function by having flow looping through T....

Example:

S-->A, cap = 4. A-->C, cap=3. C-->T, cap = 2.
S-->B, cap = 2. B-->D, cap=3. D-->T, cap = 4.
C-->B, cap = 1. B-->C, cap=2.

LP is: maximize $x_{ct} + x_{dt}$ subject to

$0 \le x_{sa} \le 4$, $0 \le x_{ac} \le 3$, 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? First solve for max flow f. Then add a constraint that flow must equal f, and minimize linear cost function $\sum_e \text{cost}(e) * x_e$. Or, you can do both together 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).

2-PLAYER ZERO-SUM GAMES:

Remember back to hwk1, where we looked at 2-player zero-sum games. Had Alice hiding a nickel or quarter 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 --- i.e., one you would want to play if you imagine that your opponent knows you well.

Another game: shooting a penalty kick against a goalie who is a bit weaker on one side. Kicker can kick left or right. Goalie can dive left or right. Here is payoff matrix for kicker (chance of getting goal):

	goalie: left	goalie: right
kicker: left	0	1

```
      kicker
             right     1     1/2
```

minimax optimal strategy for kicker: 1/3 chance kick left, 2/3 chance kick right. Expected gain = 2/3.

How about solving an nxn game?

```
            20    -10   5

            5     10    -10

            -5    0     10
```

How can we compute minimax optimal strategy (say, for row)?

Variables: the things we want to figure out are the probabilities we should put on our different choices. Those are our variables: p_1,...,p_n

Need to be legal prob dist: for all i, p_i >= 0. p_1 + ... + p_n = 1.

Want to maximize the worst case (minimum), over all columns opponent can play, of our expected gain. This is a little confusing because we are miximizing a minimum. So, add one new variable v (representing the minimum) and put in constraints that our expected gain has to be at least this, then maximize on v. So this looks like:

```
            maximize v such that:
```

p_i are a prob dist: (see constraints above).
for each column j, p_1*M[1][j] + p_2*M[2][j] + ... + p_n*M[n][j] >= v.

E.g., in example above we want:

```
get at least v if col1:   20*p_1 + 5*p_2 -5*p_3 >= v
.............. if col2:  -10*p_1 + 10*p_2 >= v
.............. if col3:    5*p_1 - 10p_2 + 10p_3 >= v
```

HOW TO SOLVE

How to solve linear programs? History: the standard algorithm for solving LPs the Simplex Algorithm, developed in the 40s. It's *not* guaranteed to run in polynomial time, and you can come up with bad examples for it, but in general it 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 is pretty slow). Later on, a faster polynomial-time algorithm Karmarkar's Alg was developed, which is competitive with Simplex. There are a lot of commercial LP packages, for instance LINDO, CPLEX, Solver (in Excel).

Won't have time to describe any of these in detail. Instead, give some intuition by viewing LP 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 <= 6 is saying that we need solution to be on a specified side of a certain hyperplane. 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 to first example with S, P, E. To make this easier to draw, can use our first constraint that S+P+E = 168 to replace S with 168-P-E. So, just draw in 2 dimensions: P and E.

Constraints are:

E >= 56
P+E >= 70
P >= 0
S >= 60 which means 168-P-E >= 60 or P+E <= 108.
2S-3P+E >= 150 which means 2(168-P-E)-3P+E >= 150 or 5P+E <= 186

For objective of max P, this happens at E=56, P = 26.
For objective of max 2P+E, this happens at P=19.5, E=88.5

Can use this view to motivate the algorithms.

SIMPLEX ALG: Earliest and most common algorithm called Simplex method. Idea: start at some "corner" of the feasible region (to make this easier, we can add in "slack variables" that will drop out when we do our optimization). Then we repeatedly do the following step: look at all neighboring corners and go to the best one if it is better. Stop when we get to a corner where no neighbor is better than we are. Neat fact is that since the objective is linear, the optimal solution will be at a corner (or maybe multiple corners). Furthermore, there are no local maxima: if you're *not* optimal then some neighbor of you is better than you are. That's because this is a convex region. So, 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.

ELLIPSOID ALGORITHM: Invented by Khachiyan in 1980 in Russia.

Solve "feasibility problem" (Then can do binary search with our objective function). Start with big ellipse (called an ellipsoid in higher dimensions) that we can be sure contains the feasible region. Then try the center of the ellipse -- see if it violates any constraints. If not, you're done. If so, look at the 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 portion of our initial ellipse. Then repeat. Can show that in each step, the new smaller ellipse has a volume that's significantly smaller -- each ellipse has volume at most (1 - 1/n) of the previous one. If ever get to too small volume can prove there can't be a solution.

One nice thing about ellipsoid algorithm is you just need to tell if the current solution violates any constraints or not, and if so, to produce one. Don't need to explicitly write them all down.

KARMARKAR'S ALGORITHM: Sort of has aspects of both. Works with feasible points but doesn't go from corner to corner. Instead it moves inside the interior of the feasible region. One of first of a whole class of so-called "interior-point methods".

Development of better and better algorithms is a big ongoing area of research. In particular, get a lot of mileage by using good data structures to speed up time in making each decision.