15-451 Algorithms
Lectures 12-14

Author: **Avrim Blum**

Instructors: **Avrim Blum**
Manuel Blum

Department of Computer Science
Carnegie Mellon University

September 28, 2011

Contents

Lecture 12

Dynamic Programming

12.1 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.

12.2 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.

12.3 Example 1: Longest Common Subsequence

Definition 12.1 *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:

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

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. 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:

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

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 correspónts 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`.

12.4 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.

12.5 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 12.2 *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.

12.6 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 12.3 *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:

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

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]

12.7 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 13

Graph Algorithms I

13.1 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).

13.2 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.

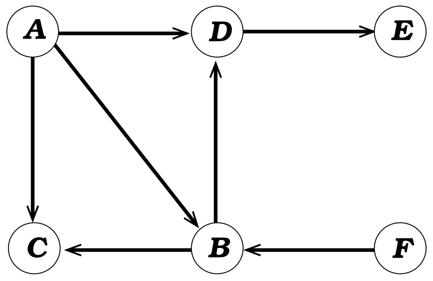
To make sure we are all on the same page, what is the maximum number of total edges in an *undirected* graph? Answer: $\binom{n}{2}$. What about a *directed* graph? Answer: $n(n-1)$.

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.

13.3 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 13.1 *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. ■

13.4 Shortest Paths

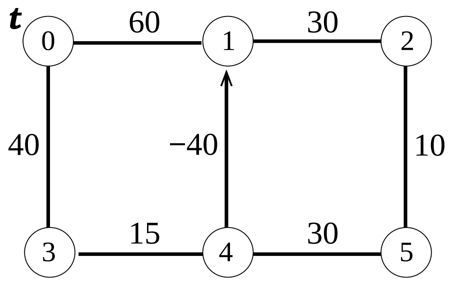
We are now going to turn to another basic graph problem: finding shortest paths in a weighted graph, and we will look at several algorithms based on Dynamic Programming. For an edge (i, j) in our graph, let's use $len(i, j)$ to denote its length. The basic shortest-path problem is as follows:

Definition 13.1 *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.

13.4.1 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.

13.5 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 $O(n^3)$.

13.5.1 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 $O(n^3 \log n)$.

13.5.2 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 14

Graph Algorithms II

14.1 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.

14.2 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 14.1 *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:

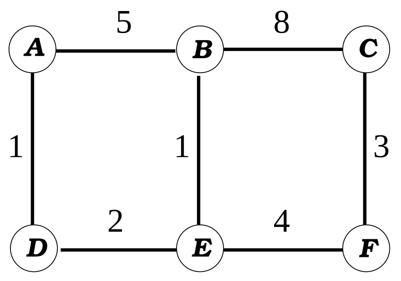
1. For each neighbor x of the tree, compute an (over)-estimate of its distance to s:

$$\text{distance}(x) = \min_{e=(v,x):v\in\text{tree}} [\text{distance}(v) + len(e)] \tag{14.1}$$

   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 (14.1)).

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



Theorem 14.1 *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).

14.3 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:

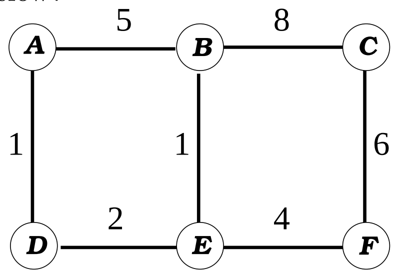
$$\text{widthto}(x) = \max_{e=(v,x):v\in\text{tree}} [\min(\text{widthto}(v), \text{width(e)})]$$

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.

14.4 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?



14.4.1 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 14.2 *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").

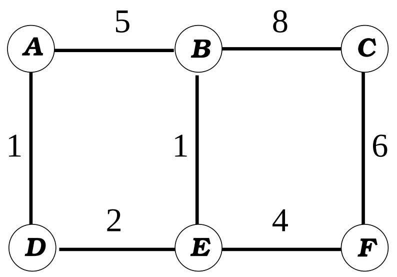
14.4.2 Kruskal's algorithm

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 14.3 *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.