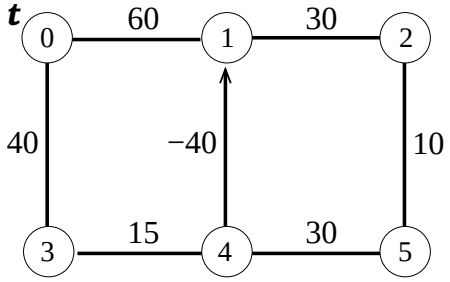
1 Dynamic Programming

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", except that unlike divide-and-conquer, the subproblems will typically overlap. In this lecture and next, we will present a few important examples.

Basic Idea: The basic idea of Dynamic Programming is to find a way to break the problem 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. As mentioned above, it is OK if our subproblems overlap. We just need the properties that (a) there are not too many subproblems overall, and (b) there is some way of ordering our subproblems such that given the solutions to the subproblems solved so far, we can fairly quickly solve the next subproblem. Then finally the last subproblem should be the original problem we wanted to solve (or something close enough to it that we can use our final subproblem to quickly solve the original problem).

2 The Bellman-Ford Algorithm

The Bellman-Ford Algorithm is a dynamic programming algorithm for the single-sink (or single-source) shortest path problem. It is slower than Dijkstra's algorithm, but can handle negative-weight directed edges, so long as there are no negative-weight cycles. 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, let's just compute the *lengths* of the shortest paths first, and afterwards we can use these lengths to easily reconstruct the paths themselves. Next, to use Dynamic Programming, we need to define subproblems. The subproblems here will be finding the shortest path from each node v to t that uses i or fewer edges, if such a path exists. More specifically, 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. Why? Because more than that will create a cycle, and we can then just cut out the cycle to get a shorter path (remember there are no negative-weight cycles).

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)∈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 $O(mn)$.

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.

3 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 $O(n^3)$.

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

3.2 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!

4 Longest Common Subsequence

[If we have time, we'll do this today, otherwise next lecture]

Definition 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:

$S =$ ABAZDC

$T =$ 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:

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

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:

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

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

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

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

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