This is an introduction to the algorithm-design technique called *dynamic programming.*

In class we discussed the concepts of worst-case running time, polynomial-time algorithms, and exponential-time algorithms. We don't cover those here.

We give some definitions, then several examples.

Definition 1. *A directed graph $G = (V, E)$ is* topologically ordered *if its vertex set V is ordered $V = (v_1, v_2, \ldots, v_n)$ and each edge (v_i, v_j) in E goes "forward" in the ordering. That is, $i < j$.*

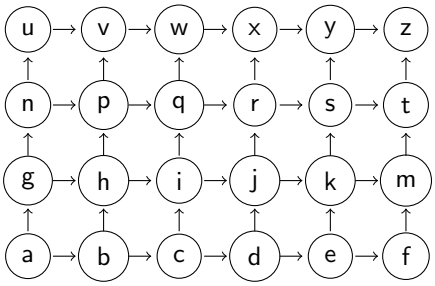
Such a graph is necessarily acyclic. That is, it is a DAG (directed acyclic graph).

Counting Paths

input: A topologically ordered DAG $G = (V, E)$, source vertex s, destination vertex t.

output: The number of paths from s to t.

For example, how many paths are there from node a to node z in the graph below?



Let P_z be the set of paths from node a to node z. We want to count its size, $|P_z|$. Here's a start. Let P_y be the set of paths from node a to node y. Let P_t be the set of paths from node a to node t.

Observation 1. $|P_z| = |P_y| + |P_t|$.

To see this, partition the set P_z of paths from node a to node z into two subsets, P_{yz} and P_{tz}, where P_{yz} contains the paths that end in edge (y, z), and P_{tz} contains the paths that end in edge (t, z). These two subsets partition P_z, so $|P_z| = |P_{yz}| + |P_{tz}|$. So we have reduced our problem to two problems: computing $|P_{yz}|$, and computing $|P_{tz}|$.

Next, note that $|P_{yz}| = |P_y|$. That is, the number of paths from node a to node z that end in edge (y, z) equals the number of paths from node a to node y. This is because there is a bijection ϕ between P_y and P_{yz}: each path p in P_{yz} (from node a to node z, ending in (y, z)) corresponds to the path $p' = \phi(p)$ in P_y obtained from p by removing the edge (y, z); conversely, each path p' in P_y corresponds to the path $p = \phi^{-1}(p')$ in P_{yz} obtained from p' by appending the edge (y, z). (For example, the path $p = (\mathsf{a}, \mathsf{g}, \mathsf{n}, \mathsf{u}, \mathsf{v}, \mathsf{w}, \mathsf{x}, \mathsf{y}, \mathsf{z})$ in P_{yz} corresponds to the path $p' = (\mathsf{a}, \mathsf{g}, \mathsf{n}, \mathsf{u}, \mathsf{v}, \mathsf{w}, \mathsf{x}, \mathsf{y})$ in P_y.) Since there is a bijection between P_{yz} and P_y, we know that $|P_{yz}| = |P_y|$.

Likewise, $|P_{tz}| = |P_t|$. It follows that $|P_z| = |P_{yz}| + |P_{tz}| = |P_y| + |P_t|$. So, if we knew $|P_y|$ (the number of paths from node a to node y), and $|P_t|$ (the number of paths from node a to node t), we could sum those two numbers to obtain the number we want ($|P_z|$, the number of paths from node a to node z). This idea gives the following recursive algorithm for Counting Paths:

1. $\mathsf{countPaths}(G, s, t)$ — return the number of paths from s to t in DAG G
2. if $s = t$, then return 1 *(base case)*
3. return $\sum_{w:(w,t)\in E} \mathsf{countPaths}(G, s, w)$ *(sum, over t's in-neighbors w, of the number of s-to-w paths)*

There is one problem with this approach. This algorithm doesn't run in polynomial time. Following the example, imagine running it on an n-vertex grid-graph. The s-to-t paths have length $2\sqrt{n}$. So the depth of the recursion tree will be $2\sqrt{n}$. And (for a grid graph) the recursive calls at depth $\sqrt{n}$ or less each make two recursive calls. So the top $\sqrt{n}$ levels of the recursion tree form a rooted binary tree of depth $\sqrt{n}$, with about $2^{\sqrt{n}}$ nodes. So the number of subproblems that is solves won't be polynomial!

But the number of distinct subproblems is polynomial. But how many *distinct* subproblems are there? When we run $\mathsf{countPaths}(G, s, t)$, each recursive call that it generates (directly or indirectly) is of the form $\mathsf{countPaths}(G, s, v)$ for some $v \in V$. That is, we recursively solve only $|V|$ subproblems, one for each vertex $v \in V$. So, the reason that the previous recursive implementation can give rise to exponentially many subproblems is that *it solves each subproblem many times.* You can see this by trying it out on the example.

There are two standard ways to fix this. One (called *memoizing*) is to cache the values from previous calls, instead of recomputing them from scratch:

0. maintain global cache array N, initially empty

countPaths$(G = (V, E), s, t)$ — return the number of paths from s to t in DAG G

1. if $s = t$, then return 1 *(base case)*
2. if $N[t]$ has not previously been set then:

3.1. set $N[t] = \sum_{w:(w,t)\in E} \mathsf{countPaths}(G, s, w)$ *(store the value in the cache)*

4. return $N[t]$

With this implementation, if a subproblem $\mathsf{countPaths}(G, s, w)$ has been previously solved, then the call will return the value in constant time. So the total time for the computation is reduced. The total time is proportional to the sum, over the distinct subproblems (one for each $w \in |V|$), of the time to solve that subproblem *the first time* (proportional to the in-degree of w). That is, the total time is proportional to $\sum_{w\in V} 1 + \mathsf{in\text{-}degree}(w)$, which is $|V| + |E|$. That is, linear in the size of the input graph G.

The alternative is to explicitly iterate over all of the subproblems, rather than recursing:

$\mathsf{countPaths}(G = (V, E), s, t)$ *(return the number of paths from s to t, assuming G is topologically sorted)*

1. assumption: the vertex set $V = (v_1, v_2, \ldots, v_n)$ is ordered so $i < j$ for each edge $(v_i, v_j) \in E$.
2. for $j = 1, 2, \ldots, n$:

3.1. set $N[v_j] = \begin{cases} 1 & \text{if } v_j = s \\ \sum_{v_i:(v_i,v_j)\in E} N[v_i] & \text{otherwise.} \end{cases}$

4. return $N[v_t]$

The iterative implementation assumes that G is topologically sorted. That way, during the iteration for any vertex v_j, for each of the vertices v_i with edges into v_j, it must be that $i < j$, so that $N[v_i]$ has already been set in an earlier iteration. The run time for this implementation is also linear in the size of G, by the same reasoning (the time is proportional to $\sum_{j=1}^{n} 1 + \mathsf{in\text{-}degree}(v_j)$, which is $|V| + |E|$).

The four-step recipe for presenting a dynamic-programming algorithm. This algorithm uses a divide-and-conquer approach to reduce the given problem several subproblems, and then

recurses to solve those subproblems. The naive recursive implementation is slow because the subproblems that arise are not distinct, the same subproblems occur (and are re-solved) many times. *Memoizing* the recursive algorithm uses a cache so that each subproblem is solved only once. Alternatively, we can use an iterative implementation, which iterates over the subproblems directly, avoiding recursion.

This is called *dynamic programming*. Here are the four steps for presenting such an algorithm:

Define the subproblems. Define the class of subproblems to be solved for a given instance, and state how the final answer is computed from these subproblems. Given a Counting Paths instance $(G = (V, E), s, t)$, we would say:

> *1. For each vertex $v \in V$, define $N[v]$ to be the number of paths from s to v. The solution to the problem is $N[t]$.*

This tells the reader exactly what subproblems the algorithm solves, and how the algorithm computes the solution from those subproblems.

State the recurrence relation. This is an identity (an equation) that captures how divide-and-conquer can be used to solve each subproblem. The left-hand side of the equation is the value of the subproblem. The right-hand side is an expression that typically contains the values of "smaller" subproblems. For Counting Paths, here is the recurrence relation:

> *2. $N[s] = 1$, and, for each $w \in V \setminus \{s\}$, we have $N[w] = \sum_{u:(u,w)\in E} N[u]$.*

The recurrence relation does not define $N[w]$. Rather, it is an identity that should hold for $N[w]$ as it is defined in the first step.

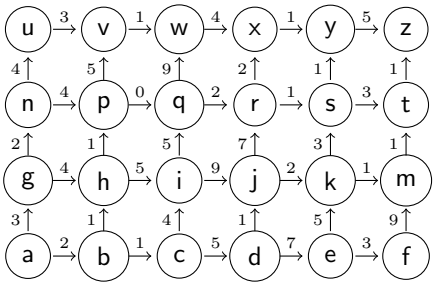
The first and second step together determine the algorithm: the algorithm will iterate (or recurse) over the set of subproblems to be solved. It will solve each subproblem using the recurrence relation (by evaluating the right-hand side to determine the value of the left-hand side). In most cases, it is not necessary to give any further description of the algorithm.

Run time. Analyze the runtime of the algorithm. This is the sum, over all of the subproblems, of the time to solve the subproblem by evaluating the right-hand side of the recurrence relation (assuming all smaller subproblems are already solved). For Counting Paths, the following would be enough:

> *3. The time to evaluate the right-hand side of the recurrence for a given vertex w is proportional to the in-degree of the vertex. So the total time for the algorithm is $\sum_{w\in V} 1 + \mathsf{in\text{-}degree}(w) = |V| + E|$. That is, it is linear in the size of the given graph.*

Correctness. To prove that the algorithm is correct, prove that the recurrence relation holds for each subproblem. The proof describes and carefully verifies the reasoning for why the recurrence relation holds. We did not do this step for Counting Paths, although we did explain the idea by example. We'll do it more carefully for later examples.

SHORTEST PATH. The reasoning underlying the COUNTING PATHS example can easily be extended to solve SHORTEST PATH. Here is the same graph, but with edge weights:



Recall the reasoning for counting the number of paths from node a to node z. Recall that P_z is the set of paths from node a to node z. Likewise P_t is the set of paths from node a to node t, and P_y is the set of paths from node a to node y. Recall that P_{tz} is the set of paths in P_z that end in edge (t,z), and P_{yz} is the set of paths in P_z that end in edge (y,z).

We observed that P_{tz} and P_{yz} partition P_z (each path to z either ends in edge (t,z), or ends in edge (y,z)). So $|P_z|$ (the quantity we want to compute) is $|P_{tz}|+|P_{yz}|$. We then observed that there is a bijection between P_{tz} and P_t (each path p ending in edge (t,z) corresponds to a path p' ending at node t, and vice versa). So $|P_{tz}| = |P_t|$. Likewise, $|P_{yz}| = |P_y|$. So $|P_z| = |P_{tz}| + |P_{yz}| = |P_t| + |P_y|$.

For SHORTEST PATH, the underlying set of objects (the set of paths) is the same. But instead of *counting* the paths in P_z, we want to find the minimum *weight* of any such path, $\mathsf{dist}(s,z)$. We can use almost the same reasoning. Any shortest path p in P_z is either in P_{tz}, or in P_{yz}. In the first case, p consists of a shortest path p' in P_t, followed by the edge (t,z). In the second case, p consists of a shortest path p' in P_y, followed by the edge (y,z).

So $\mathsf{dist}(s,z) = \min(\mathsf{dist}(a,t) + \mathsf{wt}(t,z), \mathsf{dist}(a,y) + \mathsf{wt}(y,z))$. Here is a more precise calculation:

$$
\begin{aligned}
\mathsf{dist}(s,z) &= \min\{\mathsf{wt}(p) : p \in P_z\} && \textit{(by definition of } \mathsf{dist}(s,z) \textit{ and } P_z\textit{)}\\
&= \min\{\mathsf{wt}(p) : p \in P_{tz} \cup P_{yz}\} && \textit{(because } P_z = P_{tz} \cup P_{yz}\textit{)}\\
&= \min\begin{cases}\min\{\mathsf{wt}(p) : p \in P_{tz}\}\\ \min\{\mathsf{wt}(p) : p \in P_{yz}\}\end{cases} && \textit{(because } \min(S_1 \cup S_2) = \min(\min S_1, \min S_2)\textit{)}\\
&= \min\begin{cases}\min\{\mathsf{wt}(p') + \mathsf{wt}(t,z) : p' \in P_t\}\\ \min\{\mathsf{wt}(p') + \mathsf{wt}(y,z) : p' \in P_y\}\end{cases} && \textit{(because } p' \in P_t \iff p' \cup \{(t,z)\} \in P_{tz}\\
& && \quad \textit{and } p' \in P_y \iff p' \cup \{(y,z)\} \in P_{yz}\textit{)}\\
&= \min\begin{cases}\min\{\mathsf{wt}(p') : p' \in P_t\} + \mathsf{wt}(t,z)\\ \min\{\mathsf{wt}(p') : p' \in P_y\} + \mathsf{wt}(y,z)\end{cases}\\
&= \min\begin{cases}\mathsf{dist}(a,t) + \mathsf{wt}(t,z)\\ \mathsf{dist}(a,y) + \mathsf{wt}(y,z).\end{cases}
\end{aligned}
$$

Generalizing this idea leads to the following dynamic-programming algorithm for SHORTEST PATH. Fix an input $(G = (V,E), s, t)$.

The subproblems. For each vertex $v \in V$, define $D[v]$ to be $\mathsf{dist}(s,v)$, the distance from s to v in G. The solution to the given problem (G,s,t) is $D[t]$.

The recurrence relation. $D[s] = 0$. For $w \neq s$, then, $D[w] = \min\{D[u] + \mathsf{wt}(u,w) : (u,w) \in E\}$.

(We've seen this identity before in the note on Dijkstra's algorithm.)

Run time. The time to evaluate the right-hand side of the recurrence for a given vertex w is proportional to the in-degree of w. So the total time is proportional to $\sum_{w \in V} 1 + \mathsf{in\text{-}degree}(w) = |V| + |E|$, linear in the size of the input G.

Correctness. Here we should prove that the recurrence relation is correct. We explained the idea by example, above. For a full proof, see the lecture note on Single-Source Shortest Paths and Dijkstra's algorithm.

In case it helps to see it explicitly, here is the resulting algorithm (the iterative version):

$\mathsf{shortestPath}(G = (V, E), s, t)$ *(return* $\mathsf{dist}(s, t)$*, assuming* G *is topologically sorted)*

1. assumption: the vertex set $V = (v_1, v_2, \ldots, v_n)$ is ordered so $i < j$ for each edge $(v_i, v_j) \in E$.
2. for $j = 1, 2, \ldots, n$:

3.1. set $D[v_j] = \begin{cases} 0 & \text{if } v_j = s \\ \min\{D[v_i] + \mathsf{wt}(v_i, v_j) : (v_i, v_j) \in E\} & \text{otherwise.} \end{cases}$

4. return $D[v_t]$

Note that we don't assume that the edge-weights are non-negative.

Longest Path. The same reasoning leads to a linear-time algorithm for the Longest Path problem, defined as follows: given an edge-weighted, topologically ordered DAG, and two vertices s and t, compute the maximum weight of any path from s to t in G. We leave the details as an exercise. (The algorithm and its derivation are exactly the same as for Shortest Path, but each min is replaced by a max.)

External resources on dynamic programming

- CLRS Chapter 15. Dasgupta et al. Chapter 6. Kleinberg & Tardos Chapter 6.
- Jeff Edmond's Lecture 5:
  http://jeffe.cs.illinois.edu/teaching/algorithms/notes/05-dynprog.pdf
- MIT lecture videos
  - https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-006-introduction-to-algorithms-fall-2011/lecture-videos/lecture-19-dynamic-programming-i-fibonacci-shortest-paths/
  - https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-006-introduction-to-algorithms-fall-2011/lecture-videos/lecture-20-dynamic-programming-ii-text-justification-blackjack
  - https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-006-introduction-to-algorithms-fall-2011/lecture-videos/lecture-21-dp-iii-parenthesization-edit-distance-knapsack
  - https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-006-introduction-to-algorithms-fall-2011/lecture-videos/lecture-22-dp-iv-guitar-fingering-tetris-super-mario-bros
- Seam Carving: https://en.wikipedia.org/wiki/Seam_carving, CLRS Problem 15-8.