Longest path in a directed acyclic graph (DAG)

Mumit Khan
CSE 221

April 10, 2011

The longest path problem is the problem of finding a simple path of maximal length in a graph; in other words, among all possible simple paths in the graph, the problem is to find the longest one. For an unweighted graph, it suffices to find the longest path in terms of the number of edges; for a weighted graph, one must use the edge weights instead. To explain the process of developing this algorithm, we'll first develop an algorithm for computing the single-source longest path for an unweighted directed acyclic graph (DAG), and then generalize that to compute the longest path in a DAG, both unweighted or weighted.

We've already seen how to compute the single-source shortest path in a graph, cylic or acyclic — we used BFS to compute the single-source shortest paths for an unweighted graph, and used Dijkstra (non-negative edge weights only) or Bellman-Ford (negative edge weights allowed) for a weighted graph without negative cycles. If we needed the shortest path between all pairs, we could always run the single-source shortest path algorithm using each vertex as the source; but of course there is Floyd-Warshall ready to do just that. The use of dynamic programming or a greedy algorithm is made possible by the fact the shortest path problem has optimal substructure — the subpaths in the shortest path between two vertices must themselves be the shortest ones!

When it comes to finding the longest path however, we find that the problem does not have optimal substructure, and we lose the ability to use dynamic programming (and of course greedy strategy as well!). To see that the longest path problem does not have optimal substructure, consider the graph in Fig. 1. The longest path from A to E is A–B–C–D–E.

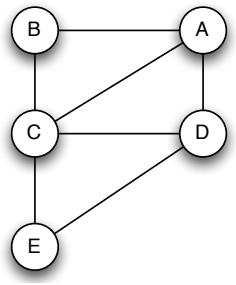

Figure 1: Optimal substructure for the longest path problem

This goes through B, but the longest path from B to E is not B–C–D–E. It's B–C–A–D–E. This problem does *not* have optimal substructure, which means that we cannot solve it using dynamic programming. Well, why not simply enumerate all the paths, and find the longest one? Because, there can be *exponentially* many such paths! For example, in the graph shown in Fig. 2, there are 2^n different paths from vertex 1 to n.[1]

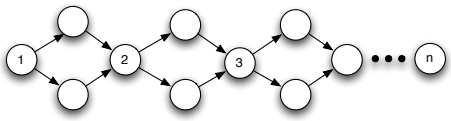

Figure 2: Number of paths in a DAG

Fortunately, the longest path in a DAG *does* have optimal substructure, which allows us to solve for it using dynamic programming (there's a trivial greedy algorithm for single-source longest path: for each vertex in the linearized order, *relax* each adjacent vertex). I will use the description used in *Algorithms* by Dasgupta, Papadimitriou and Vazirani (Ch. 4).

Let's start with the DAG G, shown in Fig. 3, as an example and see how we can develop a dynamic programming solution for the single-source longest path problem. Remember that a DAG can always be *topologically sorted* or *linearized*, which allows us to traverse the vertices in linearized order from left to right. To see how this helps, look at the linearized version of G, whose vertices, taken in the linearized order, are then: $\{S, C, A, B, D, E\}$.

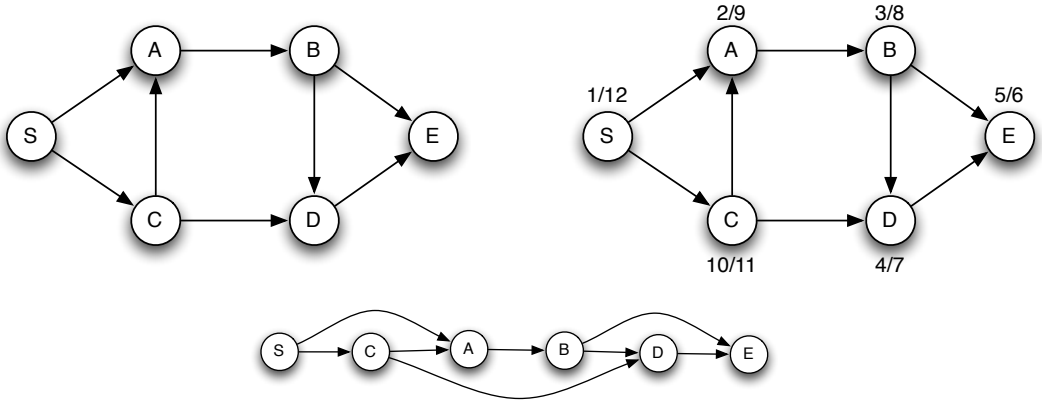

Figure 3: A DAG (top left) and its linearized version (bottom)

As I said earlier, before we tackle the problem of the finding the longest path in the graph, let's first consider the problem of finding the longest path from vertex S to any other vertex. Consider the vertex D in the linearized graph — the *only* way to get to D from S is

[1]Do you see why? Each vertex in the "top row" or the "bottom row" is either in a path or not, just like in a binary number; there are n such vertices in top or bottom row, which gives us 2^n possible paths among the $3n + 1$ vertices.

through one of its predecessors: B and C. That means, to compute the longest path from S to D, one must first compute the longest path from S to B (up to the first predecessor), and the longest path from S to C (up to the second predecessor). Once we've computed the longest paths from S to these two predecessors, we can compute the longest path to D by taking the larger of these two, and adding 1. If $\text{dist}(v)$ is the longest distance from S to vertex v, and $\alpha(v)$ is the actual path, then we can write the following recurrence for $\text{dist}(D)$:

$$\text{dist}(D) = \max\{\text{dist}(B) + 1, \text{dist}(C) + 1\}$$

Note the *subproblems* here: $\text{dist}(B)$ and $\text{dist}(C)$, both of which are "smaller" than $\text{dist}(D)$. Similarly, we can write the recurrences for $\text{dist}(B)$ and $\text{dist}(C)$ in terms of its subproblems:

$$\text{dist}(B) = \text{dist}(A) + 1$$
$$\text{dist}(C) = \text{dist}(S) + 1$$

and so on for each vertex. For a vertex with no incoming edges, we take the maximum over an empty set, which results in an expected distance of 0. Or, we can explicitly set the base case $\text{dist}(S)$ and $\alpha(S)$ for the source vertex S, as follows:,

$$\text{dist}(S) = 0$$
$$\alpha(S) = \{S\}$$

For G, by inspection, $\text{dist}(B) = 3$ and $\text{dist}(C) = 1$, which tells us that $\text{dist}(D) = \max\{3 + 1, 1 + 1\} = 4$, a fact easily verified by inspection. The corresponding actual longest path from S to D, through B, is $\alpha(D) = \alpha(B) \cup \{D\} = \{S, C, A, B, D\}$, again easily verified by inspection. We are now ready to write the recurrence for the longest path from S to any other vertex v in G, which involves first computing the longest paths to all of v's predecessors from S (these are the *subproblems*).

$$\text{dist}(v) = \max_{(\text{u,v}) \in \text{E}}\{\text{dist}(u) + 1\}$$

And we see how we can start from S, find the longest paths to all the vertices by traversing from left to right in linearized order. For a graph $G = (V, E)$, the algorithm is shown below.

for each vertex $v \in V$ in linearized order
    do $\text{dist}(v) = \max_{(\text{u,v}) \in \text{E}} \{\text{dist}(u) + 1\}$

If there are no edges into a vertex (ie., if *in-degree*$[v] = 0$), we take the maximum over an empty set, so $\text{dist}(S)$ will be 0 as expected. To implement this algorithm, we need to know the predecessor of each vertex, and the easy way to have that information to compute G^{rev}, which is G with each edge reversed. The algorithm is obviously $O(|V| + |E|)$.

Now that we've computed the longest path from S to all the other vertices $v \in V$, let's now look at our original problem of finding the longest path in G. It so happens that the

subproblems $\{\text{dist}(v),\ v \in V\}$ that we solve give us the longest path ending at some vertex $v \in V$. The longest path in G then is simply the largest of all $\{\text{dist}(v),\ v \in V\}$!

$$\text{longest-path}(G) = \max_{\text{v}\in\text{V}} \{\text{dist}(v)\}$$

And, we can compute this bottom-up for each vertex $v \in V$ taken in a linearized order. The final algorithm is shown below:

LONGEST-PATH(G)
▷ Input: Unweighted DAG $G = (V, E)$
▷ Output: Largest path cost in G
Topologically sort G
for each vertex $v \in V$ in linearized order
do $\text{dist}(v) = \max_{(\text{u,v})\in\text{E}} \{\text{dist}(u) + 1\}$
return $\max_{\text{v}\in\text{V}} \{\text{dist}(v)\}$

What would we do if we instead had a weighted DAG? Each edge would then contribute its edge weight, instead of 1, to the path cost. Fig. 4 shows an example weighted DAG G with its topologically sorted version. For a weighted graph $G = (V, E)$, and with edge

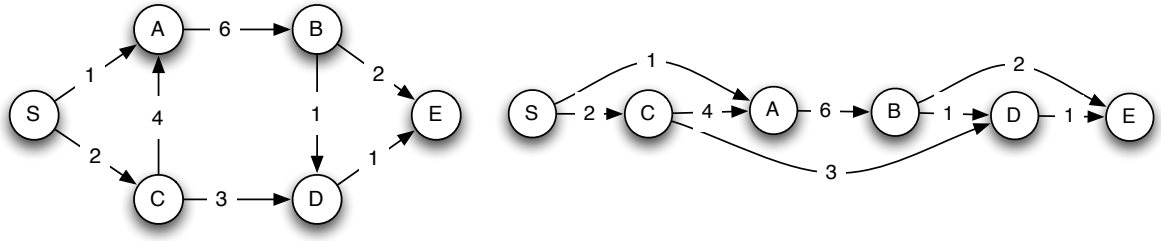

Figure 4: A weighted DAG (left) and its linearized version (right)

weights $w(u, v)$, the algorithm is then:

LONGEST-PATH(G)
▷ Input: Weighted DAG $G = (V, E)$
▷ Output: Largest path cost in G
Topologically sort G
for each vertex $v \in V$ in linearized order
do $\text{dist}(v) = \max_{(\text{u,v})\in\text{E}} \{\text{dist}(u) + w(u, v)\}$
return $\max_{\text{v}\in\text{V}} \{\text{dist}(v)\}$

The modification to return the actual path $\alpha(\cdot)$ along with $\text{dist}(\cdot)$ is left as an exercise.