Single Source Shortest and Longest Paths in a DAG

Arnab Ganguly, Assistant Professor
Department of Computer Science, University of Wisconsin – Whitewater
Data Structures (CS 223)

1 Shortest Path

An important application of topological sorting is computing the shortest path from a given vertex s (called source) to a given vertex t (called target), i.e., among all paths from s to t, find a path that has the minimum weight, the weight of a path being the sum of the edge weights on that path.

First let $v_1, v_2, \ldots, v_k$ be the only vertices that have an outgoing edge to t. Then any path from s to t must travel via one of the k vertices; in particular, the shortest path must also travel through one of these k vertices. Let $D(x, y)$ represent the length of the shortest path from x and y, and for any edge (u, v), let $weight(u, v)$ represent its weight. The main intuition is that the shortest path from s to t is the one that satisfies the following equation:

$$D(s,t) = \min\{d(s,v_1)+weight(v_1,t), weight(s,v_2)+weight(v_2,t), \ldots, weight(s,v_k)+weight(v_k,t)\}$$

In other words, the length of the shortest path from s to t can be obtained by adding the weight of the edge (v_i, t), $1 \leq i \leq k$, to the shortest path from s to v_i and then picking the minimum. This property, known as **sub-path optimality**, plays a crucial role in the following algorithm.[1]

Algorithm

- First obtain a topological ordering of the DAG
- Let $distance[\ ]$ be an array, where $distance[v]$ will ultimately store the length of a shortest path from the source vertex s to v .
- Initialize $distance[v] = \infty$ for every vertex v
- Initialize $distance[s] = 0$
- For each vertex u in the topological order starting from s, do the following:
  - for every outgoing edge e of u, do the following:
    - let v be the destination of e and $weight(u, v)$ be weight of the edge e
    - compute $len = distance[u] + weight(u, v)$
    - if $(len < distance[v])$, then update $distance[v] = len$

[1] In actual implementation, you do not need to compute a topological ordering first. You can simply carry out both the processes at the same time – shortest path computation and topological sorting. However, we stick to the method of computing topological sorting first as it gets one thing out of the way.

1.1 Simulation

The table below simulates the shortest path algorithm on the DAG in the following figure.

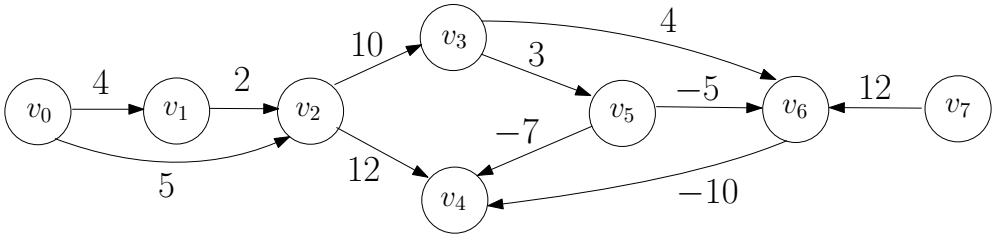

Table 1: Illustration of Shortest Path in Acyclic Graphs

Topological Order: $[v_0, v_7, v_1, v_2, v_3, v_5, v_6, v_4]$								
	distance array							
	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7
At Start:	∞	0	∞	∞	∞	∞	∞	∞
Relax v_1	∞	0	2	∞	∞	∞	∞	∞
Relax v_2	∞	0	2	12	14	∞	∞	∞
Relax v_3	∞	0	2	12	14	15	16	∞
Relax v_5	∞	0	2	12	8	15	10	∞
Relax v_6	∞	0	2	12	0	15	10	∞
Relax v_4	∞	0	2	12	0	15	10	∞

1.2 Complexity

Note that an edge or a vertex is processed only a constant number of times. Hence the complexity on a graph with N vertices and M edges is $O(N + M)$.

2 Longest Path

The key observation is that the longest s-t path in a DAG is simply the shortest s-t path in the negative DAG, i.e., the same DAG with the weights negated. We get the following algorithm.

Algorithm

- Given a DAG $\mathcal{G}$, we first obtain $\mathcal{H} = -\mathcal{G}$ by simply negating the edge weights of $\mathcal{G}$
- Now, we execute the s-t shortest path algorithm on $\mathcal{H}$ and obtain the *distance* array.
- To get the longest paths from s, simply negate the values in the *distance* array

Of course, we do not need to actually write the algorithm in this way, but simply make the following changes to the shortest path algorithm: initialize *distance* array with $-\infty$ and update $distance[v] = len$ if $len > distance[v]$. Rest of the algorithm remains the same.

Note that the longest path algorithm is the same as the shortest path algorithm with a very minor change. Hence the complexity on a graph with N vertices and M edges is still $O(N + M)$.