Shortest Paths in DAGs

Review: General Structure of Shortest Paths

- \( v.d \) - The weight of the current shortest path from \( s \) to \( v \).
- \( v.\pi \) - The parent vertex of \( v \) in the current shortest path.
- \( w(u, v) \) is the weight of the edge from vertex \( u \) to vertex \( v \)
- \( \delta(u, v) \) is the weight of the shortest path from vertex \( u \) to vertex \( v \)

Initialize:

\[
\text{for } v \in V: \quad v.d \leftarrow \infty \\
\text{for } s \in V: \quad s.d \leftarrow 0 \\
\text{v.\pi} \leftarrow \text{NIL}
\]

Main:

\[
\text{repeat} \\
\text{select edge } (u, v) \quad [\text{somehow}] \\
\text{“Relax” edge } (u, v) \\
\quad \text{if } v.d > u.d + w(u, v): \\
\quad \quad v.d \leftarrow u.d + w(u, v) \\
\quad \quad v.\pi \leftarrow u
\]

until all edges have \( v.d \leq u.d + w(u, v) \)

Shortest Paths in DAGs

Consider a directed graph which contains no cycles. We call this a directed acyclic graph (DAG). Since there are no cycles, we do not have to worry about negative weight cycles. As a result, there is a simple shortest path algorithm that works on DAG’s with arbitrary real edge weights (including negative weight edges!), which we’ll call DAG-SP.

The first step in finding the shortest path is to topologically sort the graph. Next we go through each vertex starting from the beginning, and relax each outgoing edge. This will guarantee that each edge only needs be relaxed once. A sample execution is shown in Figure 1. See a pseudocode for the procedure below.
Figure 1: Sample execution of DAG-SP. The numbers in the nodes correspond to the current distance estimates, and bold edges indicate edges that have been relaxed. To see a more complicated example, please refer to Section 24.2 in CLRS.

**DAG-SHORTEST-PATHS(G, w, s)**

1. topologically sort the vertices of G
2. Initialize distance of s to each vertex.
3. for each vertex \( v \in V[G] \)
   4. do \( d[v] \leftarrow \infty \)
   5. \( \pi[v] \leftarrow \text{NIL} \)
6. \( d[s] \leftarrow 0 \)
7. for each vertex \( u \), taken in topologically sorted order
   8. do for each vertex \( v \in Adj[u] \)
   9. do RELAX(u, v, w)

The runtime of this algorithm can be easily computed from the operations. Topologically sorting the graph takes \( O(V + E) \) time. Relaxing an edge in the graph takes \( O(1) \) time. Since each edge is relaxed only once, the total runtime of the algorithm is \( O(V + E) \).

**Proof of Correctness**

The intuition behind why relaxing each edge once is enough to find the shortest path to every node from the source node is that if a node is reachable from the source, then it must occur later in the
**topological sort.** Since we are dealing with DAGs, there are no “back-edges” from any vertex to another vertex that occurs earlier in the topological sort. If such “back-edges” exist, then it would be a contradiction to the fact that the graph is a DAG. Therefore, any path between two vertices only contain edges that go “forward” in the topological sort. Relaxing these edges in the topsort order will result in discovering and comparing the lengths of all paths from the source to all other nodes in the graph.

We prove that the algorithm is correct more formally below (taken almost verbatim from Section 24.2 in CLRS):

**Theorem 1** Given a weighted, acyclic, directed graph \( G(V, E) \) and a source \( s \), DAG-SP returns \( v.d = \delta(s, v) \) for all vertices \( v \in V \) and the predecessor subgraph (consisting of all \( v.\pi \)) is a shortest path tree at the termination of the procedure.

**Proof.** We first show that \( v.d = \delta(s, v) \) for all \( v \in V \) at the end of DAG-SP. If \( v \) is not reachable from \( s \), then \( v.d = \delta(s, v) = \infty \). Now, we suppose that \( v \) is reachable from \( s \), so that there is a shortest path \( p = \langle v_0, v_1, \ldots, v_k \rangle \), where \( v_0 = s \) and \( v_k = v \). Because we process the vertices in topologically sorted order, we relax the edges on \( p \) in the order \((v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)\). This means that \( v_i.d = \delta(s, v_i) \) at the termination of the procedure for \( i = 0, 1, \ldots, k \). The predecessors obtained from \( v_i.\pi \) will be a shortest path tree since it records the edges along the path resulting in \( v_i.d = \delta(s, v_i) \).

To see why this fails when a cycle is introduced, suppose that we introduce a negative cost cycle in the example shown in Fig. 1. We add an edge from \( t \) to \( u \) that has cost \(-5\) (see Fig. 2). The shortest path from \( s \) to \( t \) is 5 as a result from the iteration of DAG-SP. However, the shortest path that can exist can be arbitrarily small because one can traverse the negative cycle an arbitrary number of times.

**Dijkstra’s Algorithm**

The most used shortest paths algorithm for weighted directed graphs. It is sneakily the most intuitive, even though its implementation is more complicated than that of DAG-SP and BELLMAN-FORD. Imagine we drop a huge colony of ants onto the source vertex \( s \) and each goes along a single possible edge from \( s \). Whenever an ant arrives at a vertex, it splits into several different ants each following a different edge going outwards.

Each ant walks at the same speed, so when an ant reaches a vertex for the first time, we know that it will have followed the shortest path to the vertex. We then mark a vertex the first time an ant arrives with the distance the ant travels, \( \delta(s, u) \). Any other ant arriving at the same location can be obliterated, because we only really care about the first ant that arrives. This ensures that we do not check worthless paths.
Figure 2: Sample execution of DAG-SP on a graph with a negative cycle.
To turn this into an actual algorithm, we maintain a frontier of visited vertices. Each of these vertices in the frontier will have the invariant that the have been visited by ants (their shortest path has been determined from $s$). We then pick the closest vertex to $s$ and relax its edges, thereby expanding the frontier. From here, we get our familiar algorithm:

\[
\text{Dijkstra}(G, W, S) \\
1 \text{ Initialize } (G, s) \triangleright \text{ uses priority queue } Q \\
2 \quad S \leftarrow \emptyset \\
3 \quad Q \leftarrow V[G] \triangleright \text{ Insert into } Q \\
4 \quad \text{ while } Q \neq \emptyset \\
5 \quad \quad \text{ do } u \leftarrow \text{EXTRACT-MIN}(Q) \triangleright \text{ deletes } u \text{ from } Q \\
6 \quad \quad \quad S = S \cup \{u\} \\
7 \quad \quad \text{ for each vertex } v \in \text{Adj}[u] \\
8 \quad \quad \quad \text{ do } \text{RELAX}(u, v, w) \leftarrow \text{this is an implicit DECREASE\_KEY operation}
\]

In Figure 2, we see a sample execution of Dijkstra. Recall that the priority queue is used to store the shortest distances (seen so far) from the starting node $s$ to each node (seen so far). Note that DECREASE\_KEY in Dijkstra’s algorithm is used to change the shortest distance to a node if it is changed in the relaxation step.
Priority Queues

An important part of Dijkstra is the use of the priority queue, a data structure which allows us to extract the current minimum element very quickly, and also to decrease the value of any element in the data structure. This sounds awfully similar to heaps, which we learned about in this class. One of the biggest uses of heaps is actually for Dijkstra! (And other very similar graph algorithms).

In Dijkstra, we call Extract-Min once for every vertex in the graph, and for every vertex we relax its outgoing edges once, such that the total number of relaxations we do is $O(E)$. Thus, the overall runtime for Dijkstra is given by

$$O(V \cdot T_{\text{Extract-Min}} + E \cdot T_{\text{Decrease-Key}}).$$

We recall that the most common implementation of the heap is using an array, which gives us the following complexity:

- Extract-Min: $\Theta(\log V)$
- Decrease-Key: $\Theta(\log V)$

The runtime for Dijkstra using a binary min heap would then be $O(V \log V + E \log V)$.

How could we speed this up? Well turns out, binary heaps are not the most efficient heaps. Fibonacci heaps (covered in 6.854) operate in $\Theta(\log V)$ for Extract-Min and $\Theta(1)$ for Decrease-Key, in amortized time. The runtime of Dijkstra will then be $O(V \log V + E)$. We know this is an optimal implementation of a priority queue because of the $\Omega(n \log n)$ sorting bound.

Negative Weight Edges

Dijkstra is only guaranteed to work on graphs with non-negative edge weights. If a graph contains a negative weight edge, then Dijkstra may produce an incorrect distance for one or more of the nodes. Note that the edge does not need to introduce a negative weight cycle in order to break Dijkstra. In Figure 3, we see a sample execution of Dijkstra on a graph with a single negative weight edge, but no negative weight cycles.

Intuitive Proof for Dijkstra’s Algorithm Using BFS

We can provide an very broad, high-level intuitive proof for Dijkstra’s Algorithm using BFS. Consider a graph $G(V, E)$ with positive edge lengths. We “expand” the graph by replacing every edge $(u, v) \in E$ with weight $w$ with $w + 1$ vertices and path consisting of unit length edges through them. If we perform this transformation on all edges in $E$, then we obtain a new unweighted graph. Running BFS on this new graph simulates running Dijkstra’s algorithm on the original graph. We can see that Dijkstra’s algorithm “fast-forwards” the BFS to only those vertices that we care about.
(a) After initialization

(b) After relaxing edges from $s$

(c) After relaxing edges from $u$

(d) After relaxing edges from $t$

(e) After relaxing edges from $v$

**Figure 4**: Sample execution of Dijkstra's algorithm on a graph with negative weight edges. The numbers in the nodes correspond to the current distance estimates, and bold edges indicate that that edge has been relaxed. We can see that after the algorithm has terminated, the distance to node $v$ is incorrect (it should be 4).

Thus, a vertex connected by a path of smaller weight to the starting vertex will be reached by the BFS on the “expanded” graph sooner than a vertex connected by a path of larger weight. Therefore, it makes sense to continue the BFS from that vertex first since it would be the first reached by BFS.

Note that this intuitive explanation does not account for non-integer edge weights. To see the formal proof of Dijkstra’s algorithm and its correctness, please refer to Chapter 24.3.

**Comparison of Shortest Path Algorithm Runtimes**

- **Unweighted Graph** - BFS: $O(V + E)$ worst case
- **DAG** - DAG-SP: $O(V + E)$ worst case
- **Weighted Graph (Non-negative weights)** - Dijkstra: $O(V \log V + E \log V)$ worst case. $O(V \log V + E)$ amortized.
- **General Graph** - Bellman-Ford: To be continued...
Graph Transformation and Other Shortest Path Example Problems

Fall 2009 Quiz 2, Problem 5. Consider a road network modelled as a weighted undirected graph $G$ with positive edge weights where edges represent roads connecting cities in $G$. However, some roads are known to be very rough, and while traversing from city $s$ to $t$ we never want to take a route that takes more than a single rough road. Assume a boolean attribute $r[e]$ for each edge $e$ which indicates if $e$ is rough or not. Give an efficient algorithm to compute the shortest distance between two cities $s$ and $t$ that doesn’t traverse more than a single rough road. (Hint: Transform $G$ and use a standard shortest-path algorithm as a black-box.)

Solution: For each vertex $v \in G$, construct a pair of vertices $v_{\text{smooth}}$ and $v_{\text{rough}}$. For each smooth edge $e = (u, v)$, add a directed edge from $u_{\text{smooth}}$ to $v_{\text{smooth}}$ and a directed edge from $u_{\text{rough}}$ to $v_{\text{rough}}$. For each rough edge $e = (u, v)$, add a directed edge from $u_{\text{smooth}}$ to $v_{\text{rough}}$. Then generate all shortest paths rooted at $s_{\text{smooth}}$, and pick the shorter of $d[s_{\text{smooth}}, t_{\text{smooth}}]$ and $d[s_{\text{smooth}}, t_{\text{rough}}]$.

Spring 2011 Final, Problem 6. Consider a connected weighted directed graph $G = (V, E, w)$. Define the fatness of a path $P$ to be the maximum weight of any edge in $P$. Give an efficient algorithm that, given such a graph and two vertices $u, v \in V$, finds the minimum possible fatness of a path from $u$ to $v$ in $G$.

Solution: To solve this problem, it is sufficient to update the standard RELAX method so that instead of summing edge weights, we take the maximum of the two edge weights. Then we can use Dijkstra’s to compute shortest paths.