Shortest Path of DAGs

Let $G = (V, E)$ be a DAG with possibly negative edge length $c(e)$ for each $e \in E$. We want to find the distance from a given source s to each vertex $v \in V$. Note that G does not contain negative cycles simply because G does not contain any cycle. We can design a simplified version of Bellman-Ford algorithm to solve this problem. In particular, we only need to do one round of "update" (instead of $|V|-1$ rounds as in Bellman-Ford algorithm). But, the edges cannot be updated in a arbitrary order. In fact, we first need to find a linearization of G (recall that a directed graph can be linearized if and only if it is a DAG), and then update the in-edges of vertices following this linearization.

```
Algorithm BF-DAG (DAG G = (V,E), l(e) for any e ∈ E, s ∈ V)
    init array dist of size |V|: dist[s] = 0; dist[v] = ∞ for any v ≠ s;
    calculate a linearization of G;
    for v ∈ V following the order of linearization
        for each (u,v) ∈ E;
            update(u,v);
        end for;
    end for;
    for each v ∈ V: report: distance(s,v) = dist[v];
end algorithm;
```

The correctness of above algorithm for DAGs can also be proved in the same way as in the Bellman-Ford algorithm. For any vertex v, let $p = s \to v_1 \to v_2 \to \cdots \to v$ be one shortest path from s to v; we have proved that (for Bellman-Ford algorithm), if the run of the algorithm contains a sequence of updates that sequentially update the edges in this shortest path p, then $dist[v]$ will be equal to $distance(s,v)$ after these update procedures. The above algorithm update the in-edges of vertices sequentially following a linearization X. By the definition of linearization, the list of vertices of any path, including p, must be a *subsequence* of X (i.e., s is before v_1 in X, v_1 is before v_2 in X, etc). Therefore, edges in p must be updated sequentially by the above algorithm, i.e., edge (s, v_1) gets updated when processing v_1, edge (v_1, v_2) gets updated when processing v_2, and so on. See Figure 1 for an example.

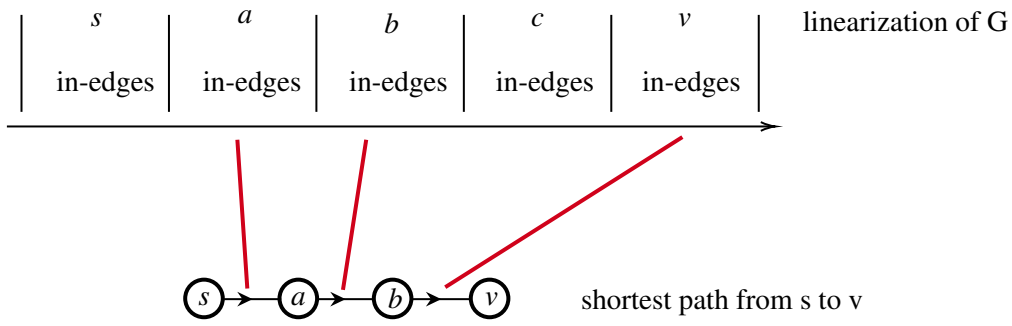

Figure 1: Illustration of the correctness of the above algorithm. Assume $s \to a \to b \to v$ be one shortest path from s to v in G. Then in any linearization of G, s is before a, a is before b, and b is before v. So, edge (s,a) will be updated when processing the in-edges of a, (a,b) will be updated when processing the in-edges of b, and (b,v) will be updated when processing the in-edges of v. Therefore, there exists a sequence of updates that sequentially update these 3 edge.

Shortest Path Tree

So far we have designed three algorithms, BFS, Dijkstra's algorithm, and Bellman-Ford algorithm to solve shortest path problems for unit edge length, positive edge length, and possible negative edge length. In these algorithm, $dist[v]$ will give the *length* of the shortest path from s to v. How to find the actual shortest path? Before designing algorithm to find paths (we will do it by modifying all three algorithm), let's think about how to store them first. Recall that we want to store $|V|$ paths, ones from s to each vertex in V. If we explicitly store these $|V|$ paths in a naive way, it may take $O(|V|^2)$ space. Can we do better?

In fact, we can use linear space to store these $|V|$ shortest paths from a single source s. How is that possible? The reason is the *optimal substructure* property that the shortest path problem satisfies. Assume $s = v_0 \to v_1 \to v_2 \cdots v_k$ be the shortest path from s to v_k. Then $s \to v_i$, for any $0 \le i < k$, is also the one shortest path from s to v_i. This implies that storing the path from s to v_k is enough to represent the shortest paths from s to each v_i.

But one such path is not enough to represent all shortest paths from s. In fact, all shortest paths can be represented as a tree, called *shortest path tree*. Without loss of generality, assume that source s can reach all vertices in V. We say T is a *shortest path tree* of G w.r.t. source vertex s if:

1. T is a rooted tree and its root is s;
2. vertices of T is V;
3. edges of T is a subset of E;
4. for every $v \in V$, the unique path from s to v in T is one shortest path from s to v in G.

We will show later that, a shortest path tree always exists. Such tree may not be unique though (see Figure 2). As in the shortest path tree, each vertex, except s, has exactly one in-edge, we can use an array *prev* of size $|V|$ to store this tree. Array *prev* is indexed by the vertices, and for each $v \in V$, $prev[v]$ stores the parent of v in the tree (see Figure 2). Such array, which takes linear space, therefore represents the shortest path from s to every vertex.

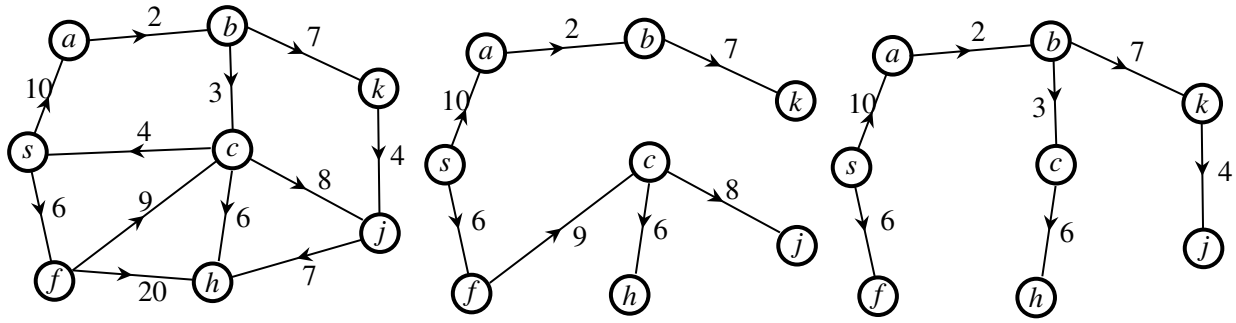

Figure 2: A directed graph and its two shortest path trees. The array representation for the first tree is $prev[s,a,b,c,f,h,j,k] = [null,s,a,f,s,c,c,b]$.

We now modify the three algorithms, BFS, Dijkstra's algorithm, and Bellman-Ford algorithm, to allow them to generate a shortest path tree. The idea is that, whenever $dist[v]$ gets updated as $dist[u] + l(u,v)$ we also immediately assign $prev[v]$ as u. See their pseudo-codes below.

```
Algorithm BFS (G = (V,E), s ∈ V)
    dist[v] = ∞, for any v ∈ V;
    prev[v] = null, for any v ∈ V;
    init an empty queue Q;
    dist[s] = 0;
    insert (Q,s);
    while (empty (Q) = false)
        u = find-earliest (Q);
        delete-earliest (Q);
        for each edge (u,v) ∈ E
            if (dist[v] = ∞)
                dist[v] = dist[u] + 1;
                prev[v] = u;
                insert (Q, v);
            end if;
        end for;
    end while;
end algorithm;
```

```
Algorithm Dijkstra (G = (V,E), l(e) for any e ∈ E, s ∈ V)
    dist[v] = ∞, for any v ∈ V;
    prev[v] = null, for any v ∈ V;
    init an empty priority queue PQ;
    for any v ∈ V: insert (PQ, v), where the priority of v is ∞;
    dist[s] = 0;
    decrease-key (PQ,s,0);
    while (empty (PQ) = false)
        u = find-min (PQ);
        delete-min (PQ);
        for each edge (u,v) ∈ E
            if (dist[v] > dist[u] + l(u,v))
                dist[v] = dist[u] + l(u,v);
                prev[v] = u;
                decrease-key (PQ, v, dist[v]);
            end if;
        end for;
    end while;
end algorithm;
```

```
Algorithm Bellman-Ford (G = (V,E), l(e) for any e ∈ E, s ∈ V)
    init an array dist of size |V|;
    dist[s] = 0; dist[v] = ∞ for any v ≠ s;
    prev[v] = null, for any v ∈ V;
    for k = 1 → |V| − 1
        for each edge (u,v) ∈ E
            update(u,v);
        end for;
    end for;
end algorithm;

procedure update (edge (u,v) ∈ E)
    if (dist[v] > dist[u] + l(u,v))
        dist[v] = dist[u] + l(u,v);
        prev[v] = u;
    end if;
end procedure;
```

The resulting *prev* array corresponds to a graph called T, in which the vertices are V and edges are $\{(prev[v], v) \mid v \in V\}$. We now prove that this graph is indeed one shortest path tree, by showing T satisfying the 4 conditions in the definition. It is trivial to verify that the conditions (2) is satisfied, as the vertices are V; condition (3) is easy as well, as in any of the algorithm, assigning u to $prev[v]$ already happens when examining edge $(u,v) \in E$. Below we then prove condition (1) and (4).

Fact 1. At any time of the algorithm, for any $v \in V$, let $u = prev[v]$, if $u \neq null$ then $dist[v] = dist[u] + l(u,v)$; in particular, when the algorithm terminates, we have $distance(s,v) = distance(s,u) + l(u,v)$.

The first part of above fact is a direct consequence of each algorithm as we update *prev* right after updating $dist[v]$ as $dist[u] + l(u,v)$. The secon part also uses the correctness of each algorithm, i.e., when each algorithm terminates, we have $dist[v] = distance(s,v)$.

We now prove condition (1), i.e., T is a tree with root being s (although we use T but we have not showed that it is a tree). Since T corresponds to array *prev*, it is obvious that in-degree of s is 0 and that each vertex in T, except s, has in-degree of 1. We therefore just need to prove that T does not contain cycle. We need a stronger assumption to make it true. That is, the graph G cannot contain negative nor length-zero cycles.

Fact 2. T does not contain cycle, if all cycles in G are of positive length.

Proof. Suppose conversely that there exists a cycle $C = v_1 \to v_2 \to \cdots \to v_k \to v_1$ in T. Following above fact, we have $dist[v_i] = dist[v_{i-1}] + l(v_{i-1}, v_i)$, for every $1 < i \leq k$ and $dist[v_1] = dist[v_k] + l(v_k, v_1)$. Summing up both sides gives $\sum_{e \in C} l(e) = 0$, contradicting to the the assumption that all cycles have positive length. □

Last, we prove condition (4):

Claim 1. For every $v \in V$, the unique path from s to v in T is one shortest path from s to v in G, if all cycles in G are positive cycles.

Proof. Let $p : s = v_0 \to v_1 \to v_2 \to \cdots \to v_k$ be the path from s to v_k in T. By the construction of T, we know that $prev[v_i] = v_{i-1}$, $1 \leq i \leq k$. Therefore, using Fact 1, we have $distance(s,v_i) = distance(s,v_{i-1}) +$

$l(v_{i-1}, v)$, for each $i = 1, 2, \cdots, k$. We sum up the left side and the right side of these k equations. Canceling out $distance(s, v_i)$, $i = 1, 2, \cdots, k-1$, on both sides gives $distance(s, v_k) = distance(s, s) + \sum_{i=1}^{k} l(v_{i-1}, v_i) = \sum_{i=1}^{k} l(v_{i-1}, v_i)$, implying that the length of p is equal to the distance from s to v_k, i.e., it is one shortest path from s to v_k. □