Single Source Shortest Paths

Shortest path:

An optimization problem, that given a graph $G = (V, E)$ and a weight function $w: E \to \mathbb{R}$, find a path $\pi =< v_0, \dots, v_l >$ - a sequence such that $\forall 0 \leq 1 \leq l - 1: (v_i, v_{i+1}) \in E$ (no loops simple path) such that $w(\pi) = \sum_{e \in \pi} w(e)$ is minimal, that is: $\pi^*(u, v) = \operatorname{argmin}_{\pi: u \to v} w(\pi)$ – the path from u to v with minimal weight, among all possible paths from u to v.

Notation:

- $\delta(u, v) = w\big(\pi^*(u, v)\big)$ – the weight of the shortest path between u and v.
- $\Delta G = A_{n \times n}$ such that $(a_{ij}) = \delta(v_i, v_j)$

$\Delta(G)$ viewed as a distance function is a metric function. Meaning:

1. $\forall v \in V(G): \delta(v, v) = 0$
2. $\forall u, v \in V(G): \delta(u, v) = \delta(v, u)$
3. The triangle inequality: $\forall u, v, w \in V(G): \delta(u, v) \leq \delta(u, w) + \delta(w, v)$

Applications:

- Routing problems: given a set of nodes and latency between them, find a shortest least delayed path to transmit from one node to another.
- Robot movement: given a set of obstacles in a space and a robot that need to get from one point to another avoiding the obstacles, find the shortest path for that.

So $\Delta(G)$ is an important matrix.

Variations of the SP problem:

- Single destination shortest path: a whole column in the matrix.
- Single pair shortest path: an entry in the matrix.
- All sources shortest paths: the whole matrix.

Is MST sufficient to compute SPs?

No. for instance, if we have a graph $v_1 \to v_2 \to \cdots v_{100} \to v_1$ with all edges but the last weighted 1, and the last weighted 2. The MST is all the 1-weight edges, but the shortest path $v_1 \to v_{100}$ is the edge $w(v_{100}, v_1) = 2$, not $w(v_1 \to \cdots \to v_{100}) = 99$.

Non-uniqueness of shortest paths:

There could be more than one shortest path in a graph.

Some properties:

- Say we have a shortest path $v_1 \to v_2 \to \cdots \to v_i \to \cdots \to v_j \to \cdots \to v_l$, then $v_i \to \cdots \to v_j$ is the shortest path from v_i to v_j. This is easily proven by contradiction.
- Given $v_i \to \cdots \to v_j \to \cdots \to v_k$ where $v_i \to \cdots \to v_j$, $v_j \to \cdots \to v_k$ are SPs, that doesn't mean $v_i \to \cdots \to v_k$ is a shortest path. However the triangle-IE applies: $\delta(v_i, v_k) \leq \delta\big(v_i, v_j\big) + \delta\big(v_j, v_k\big)$. The $\leq$ will become $=$ iff v_j is on some shortest path between v_i, v_j.

Basic operation:

Start with an initial estimate $d[u]$ that denotes the current value of the shortest path known from a source s to all other $u \in V$. The reduction of the weight of the edges is called relaxation.

For instance, say we have paths $s \rightarrow u, s \rightarrow v$ and an edge (u, v). We currently hold $d[u], d[v]$ with some estimate values (even ∞). If the case is $d[v] > d[u] + w(u, v)$ then we change it to $d[v] \coloneqq d[u] + w(u, v)$ – we got a better estimate of the shortest path. Formally:

$Relax(u, v, w)$:

$\quad if\ d[v] > d[u] + w(u, v)\ then$

$\qquad d[v] \coloneqq d[u] + w(u, v)$

end

How many times we need to relax? If we apply relaxation over and over lots of times, we would get shortest paths values in all d fields eventually, but when.

The initial values are set:

- $d[s] = 0$
- $d[u] = \infty, \forall u \neq s$

Bellman-Ford algorithm:

The algorithm is just doing the above, but stating that after $n = |V|$ runs of relaxing all $e \in E$, we get the shortest paths. After finishing it, we do one more run on all $e \in E$. If $d[v] > d[u] + w(u, v)$, it means we have a **negative cycle**. This algorithm is $\Theta(nm) = O(n^3)$.

Negative cycles:

Assume we have a path $v_0 \rightarrow \cdots \rightarrow v_i \rightarrow \cdots \rightarrow v_j \rightarrow \cdots \rightarrow v_l$ such that there's a negative cycle between v_i, v_j. That means the more we do cycles in our path, we reduce the weight of the path.

Therefore, if after n iterations we don't have all shortest paths it means the graph contains a negative cycle.

Why n times:

Given $v_{0_0} \rightarrow v_{1_\infty} \rightarrow \cdots \rightarrow v_{k_\infty}$ where v_0 is the source. We can't only have only one relaxation (round of relaxations) since we don't know the order of relaxation of the edges. For instance we could relax $(v_{k-1}, v_k), \dots, (v_0, v_1)$ and that's a relaxation round that got us only one improvement – the value of v_1 is changed.

Therefore, at most k are relaxations are needed, and $k \leq n(-1)$ so at most we would need n. Except if there's a negative cycle.

Dijkstra's algorithm

Assumes all edge weights are non-negative.

Initialization:

- Initialization: the same as BF: $d[s] = 0, d[u] = \infty \forall s \neq u \in V$
- Use a priority queue, initialized to $Q = V$

The top of the Q is the next node to be taken care of. Every $extract - min$ we update all d-values of all adjacent nodes to the node extracted.

$u = extract - \min(Q)$
$S = S \cup \{u\}$
$for\ each\ v \in adj[u]$:
$Relax(u, v, w)$

Note that the only difference between this algorithm and Prim's MST algorithm is the d-value: at Prim's it's the weight of one edge, here it's the total weight of the path.

Running time:

Initialization is n; building Q based on d-values is linear. As long as we have values is Q we need to $extract - min$ which costs $\lg n$ (if using a min-heap). Updating the d-value for each $v \in adj[u]$, across all u would derives m operations, that w.c. will conclude with a $decrease - key$ for v which means $O(\lg n)$.

That concludes to $O\big((n+m)\lg n\big)$.

Why the top of Q holds the δ:

At the time of $extract - min$ of some node $u \in Q$, we know that $d[u] = \delta(s, u)$. The reason is that all earlier extract-mins got all smallest paths, and there is no negative weight that could make any other node down the queue be closer to s than the one extracted now.

Shortest paths in DAG

A directed acyclic graph is a graph with no cycles. These graphs don't have negative cycles. Using DFS we can apply a topological sort on the graph, meaning it would take $O(m + n)$.

The algorithm:

- Initialize all d-values to ∞ except s
- For each $u \in V$ taken in topological order:
  - For each $v \in adj[u]$: $Relax(u, v, w)$

Running time: $O(DFS + \sum_{u \in V} \deg(u)) = O(n + m)$.

Because there are no cycles, once a d-value is set, we know it cannot be changed again.