Shortest Paths in Weighted Graphs

In **weighted graphs**, the length of a path is the sum of the weights of its edges.

> $\delta(u, v)$ = the length of a shortest path from u to v. If no path exists, it's set to ∞.

The **single-source shortest-path problem** involves finding $\delta(s, v)$ (and a specific path of this length) for all $v \in V$, given $s \in V$.

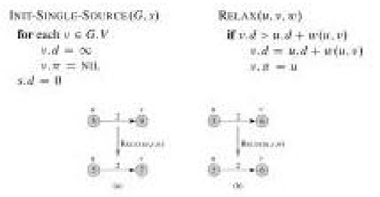
Note that prefixes of shortest paths must themselves be shortest paths: if $v_1, v_2, \dots, v_k$ is a shortest path from v_1 to v_k, then $v_1, v_2, \dots, v_i$ is a shortest path from v_1 to v_i for all $i \leq k$.

Negative Cycles

The problem is ill-defined if there are cycles (reachable from s) with a negative sum. In such cases, paths with arbitrarily low lengths exist. Conversely, if no such negative cycles exist, we only need to consider simple paths (without node repetitions). There is a finite number of such paths (at most $n!$), so the "length of the shortest path" is well-defined.

Relaxation: A General Technique

Relaxation uses edges to propagate information about known path lengths from node to node. If u has information indicating a path from s to u of length $u.d$, and (u, v) is an edge with weight w, then there exists a path of length $u.d + w$ to v. The algorithm determines whether this is better information for v than what it currently has.



```
INIT-SINGLE-SOURCE(G, s)
  for each $v \in G.V$
      $v.d = \infty$
      $v.\pi = $ NIL
  $s.d = 0$
```

```
RELAX(u, v, w)
  if $v.d > u.d + w(u, v)$
      $v.d = u.d + w(u, v)$
      $v.\pi = u$
```

The image above depicts the INIT-SINGLE-SOURCE algorithm, which initializes the distance to the source node as 0 and all other nodes as infinity, and the RELAX algorithm, which updates the distance to a node if a shorter path is found through another node.

When implementing "∞", ensure it functions mathematically correct with ">" and "+". Using `Integer.MAX_VALUE` as "∞" is insufficient.

Invariant

Algorithms start with Init-Single-Source and then only modify $v.d$ and $v.\pi$ via Relax. For such algorithms, the following invariant holds (easily proven by induction on the number of Relax operations): If $v.d < \infty$, there exists a path from s to v of length $v.d$. Therefore, $\delta(s,v) \leq v.d$ always holds.

Since $v.d$ can only decrease with Relax, if $\delta(s,v) = v.d$ at some point, $v.d$ (and thus $v.\pi$) cannot be changed later.

Specifically, if $\delta(s,v) = v.d$ for all nodes, no edge (u,v) can be relaxed (i.e., $v.d \leq u.d + w(u,v)$ holds for all edges (u,v)).

Finding Shortest Paths via $v.\pi$-Pointers

The set S of nodes v with $\delta(s,v) = v.d < \infty$ forms a tree with $v.\pi$ as parent pointers and s as the root. For a node v in the tree, the path towards the root corresponds to a backward traversal of a path in the graph from s to v of length $\delta(s,v)$.

This is shown by induction on the number of Relax operations.

- **Basis:** Immediately after initialization, s is the only node v with $v.d < \infty$. Since $s.d = 0$ and $s.\pi = \text{NIL}$ after initialization, $S = s$ and the invariant holds with a tree of size one, provided $\delta(s,s) = 0$. Note that $\delta(s,s) \neq 0$ is only possible if s lies on a negative cycle, in which case for all nodes v either $\delta(s,v) = -\infty$ (if v is reachable from s) or $\delta(s,v) = \infty$ (if v is not reachable from s). If $v.d < \infty$, then $v.d$ is the length of a specific path and is therefore different from $-\infty$.

- **Inductive Step:** For a Relax operation that changes nothing, there is nothing to show. For a Relax operation that changes $v.d$ (and thus $v.\pi$): here v cannot be in S before Relax (since $\delta(s,v) < v.d$ at that point), so all existing nodes in the tree are unchanged (including their parent pointers). If v is incorporated into S due to this Relax, let (u,v) be the edge that was relaxed, and let w be its weight. We then have $\delta(s,v) = v.d = u.d + w$. Therefore, $\delta(s,u) = u.d$ must hold (if there were a path shorter than $u.d$ to u, there would be a path shorter than $u.d + w$ to v) and $u.d < \infty$ also holds (otherwise Relax would not occur). So u is in S, gets v as a child, and the theorem clearly holds again.

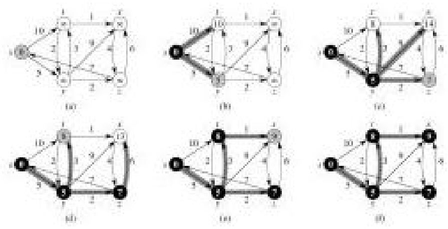
Dijkstra's Algorithm [1959]

Dijkstra's algorithm is a greedy algorithm that incrementally builds the set S of nodes with correct $v.d$ and $v.\pi$ values. It uses a priority queue Q and requires all edge weights to be non-negative (≥ 0).

- **Runtime:** n Insert (or one Build-Heap), n Extract-Min, and m Decrease-Key (in Relax). In total, $O(m \log n)$ if the priority queue is implemented with a heap.
- **Invariant:** When u is incorporated into S (i.e., extracted with an Extract-Min), $u.d = \delta(s, u)$ (if all edge weights are ≥ 0).

The correctness of the algorithm follows from the invariant (since all nodes are in S at the end).

Path-Relaxation Lemma



Now, we consider algorithms that can handle negative weights (but not negative cycles). We start with the following lemma:

Lemma: If $s = v_1, v_2, \ldots, v_k = v$ is a shortest path from s to v, and an algorithm performs Relax operations on the edges $(v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v_k)$ in turn (with any number of Relax operations of other edges between these Relax operations), then $\delta(s, v) = v.d$ after the last of these Relax operations.

Proof: It is shown by induction on i that after Relax is performed on the edge (v_{i-1}, v_i) in the sequence above, $v_i.d$ can be at most equal to the sum of the weights of the first $i - 1$ edges in the path. So after the last of these Relax operations, $v.d \leq \delta(s, v)$, since the path is a shortest path to v. Since $\delta(s, v) \leq v.d$ always holds, $\delta(s, v) = v.d$.

Algorithm for DAGs

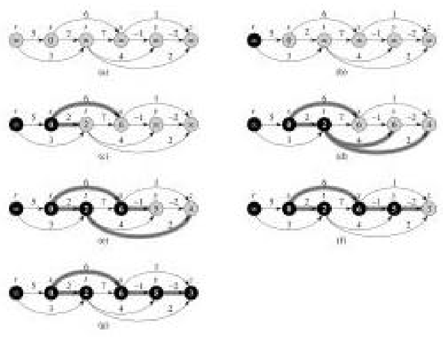
> **DAG** = Directed Acyclic Graph.

A topological sorting can be found via DFS in time $O(n + m)$.

- **Runtime:** $O(n + m)$.
- **Theorem:** When the algorithm stops, $v.d = \delta(s, v)$ for all $v \in V$.

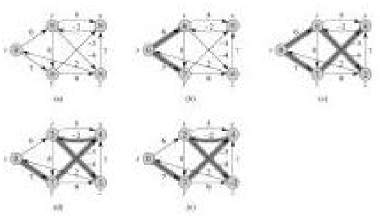
Proof: For a node v with a path from s to v: all nodes on a shortest path have been relaxed in order (whereby correct δ-values are set on this path due to the path-relaxation lemma). For all other nodes, $\infty = \delta(s, v)$, so correctness here follows from $\delta(s, v) \leq v.d$.

Example of the Algorithm for DAG



Bellman-Ford-Moore [1956-57-58]

- **Runtime:** $O(nm)$



Relaxation order: (t,x), (t,y), (t,z), (x,t), (y,x), (y,z), (z,x), (z,s), (s,t), (s,y).

The idea behind Bellman-Ford is that it maintains the following invariant: After i iterations of the first for-loop, $v.d = \delta(s,v)$ holds for all nodes v that have a shortest path with at most i edges.

This invariant follows directly from the path-relaxation lemma.

Correctness

Theorem: If there exists a negative cycle that can be reached from s, Bellman-Ford returns FALSE. Otherwise, it returns TRUE, and $v.d = \delta(s,v)$ for all $v \in V$ when it stops.

Proof:

- Case 1: There are no negative cycles reachable from s. Then all nodes reachable from s have a simple shortest path (a path without node repetitions). Such a path has at most n nodes, and therefore at most $n-1$ edges. By the invariant above, $v.d = \delta(s,v)$ holds for these nodes when the first for-loop ends. For nodes that cannot be reached from s, this already holds after initialization at the beginning. So $v.d = \delta(s,v)$ for all nodes when the first for-loop ends. When $v.d = \delta(s,v)$ for all nodes, Relax cannot change any $v.d$ values anymore. Therefore, the if-case in the second for-loop never becomes true, and the algorithm returns TRUE.

- Case 2: There is a negative cycle C that can be reached from s. We first note that if a node v can be reached from s, it can also be reached via a simple path (a path without repetitions among nodes). Such a path has at most n nodes. It is easy to see via induction over i that after i iterations of the first for-loop, the d-value is finite for the first $i+1$ nodes on this simple path. Therefore, $v.d < \infty$ holds at the end of the for-loop. Specifically, this holds for all nodes on C (they can all be reached from s). Assume that Bellman-Ford in Case 2 does not return FALSE. Then at the end of the algorithm:

$$v_{i+1}.d \leq v_i.d + w(v_i, v_{i+1}) \text{ for } 1 \leq i \leq k \text{ (with } v_{k+1} = v_1\text{).}$$

And thus:

$$\sum_{i=1}^{k} v_i.d \leq \sum_{i=1}^{k} v_i.d + \sum_{i=1}^{k} w(v_i, v_{i+1})$$

Since $v_i.d < \infty$ for all i, the first two sums are not just equal, but also $< \infty$, so they can be subtracted and give:

$$0 \leq \sum_{i=1}^{k} w(v_i, v_{i+1})$$

in contradiction with the cycle being negative. So the algorithm must return FALSE. □

Shortest Paths Between All Pairs of Nodes

The all-pairs shortest-path problem: For all $s \in V$, find $\delta(s,v)$ (and a specific path) for all $v \in V$.

- One option: run Dijkstra from each source $s \in V$ (requires non-negative weights): $O(nm \log n)$ time.
- Or: run Bellman-Ford-Moore from each source $s \in V$ (if there are negative weights): $O(n^2 m)$ time.
- Another option: Floyd-Warshall's algorithm. $O(n^3)$ time. Handles negative weights.
- Yet another option: Johnson's algorithm. Runs in $O(nm \log n)$ time. Handles negative weights.

Floyd-Warshall Algorithm [1962]

Floyd-Warshall does not use Relax; instead, it's based on dynamic programming. The input is the graph in the adjacency-matrix representation in a variant W with weights on edges: $w_{ii} = 0$, $w_{ij} = w(i, j)$ if $(i, j) \in E$, $w_{ij} = \infty$ otherwise. The output is also in matrix form: $D = (d_{ij})$, $d_{ij} = \delta(v_i, v_j)$ = the length of a shortest path from v_i to v_j. Set to ∞ if no path exists. $\Pi = (\pi_{ij})$, π_{ij} = last node before v_j on a shortest path from node v_i to node v_j. Set to NIL if no path exists.

(Only construction of the D-matrix is shown; see the book for the Π-matrix.)

- **Runtime:** $O(n^3)$.
- **Space:** $O(n^2)$ (only the previous $D^{(k)}$ matrix needs to be stored).
- **Theorem:** When the algorithm stops, d_{ij} and π_{ij} in the last matrix are set correctly for all $v_i, v_j \in V$ (if no negative cycle is in the graph).

Proof: The invariant is that $D^{(k)}$ contains the length of the shortest path between v_i and v_j that only passes through the nodes $v_1, v_2, \ldots, v_k$ (besides the endpoints v_i and v_j). This is shown by induction on k.

Johnson's Algorithm [1977]

Johnson's Algorithm uses:

- Bellman-Ford-Moore once on a slightly extended graph.
- From this, adjustment of edge weights so that all become positive without essentially changing shortest paths (see lemma on the next page).
- Runs Dijkstra from all nodes.

It runs in $O(nm \log n + nm) = O(nm \log n)$ time and handles negative weights.

Re-weighting

Consider the situation where we assign a number $\phi(v)$ to all nodes $v \in V$. Based on ϕ, we can create new weights $\widetilde{w}$ in the graph in the following way:

$$\widetilde{w}(u, v) = w(u, v) + \phi(u) - \phi(v).$$

Consider a path $v_1, v_2, \ldots, v_k$. Then:

$$\sum_{i=1}^{k-1} \widetilde{w}(v_i, v_{i+1}) = \sum_{i=1}^{k-1} (w(v_i, v_{i+1}) + \phi(v_i) - \phi(v_{i+1})) = \sum_{i=1}^{k-1} w(v_i, v_{i+1}) + (\phi(v_1) - \phi(v_k))$$

since the sum is telescoping.

In other words, the path length under the new weights is equal to the path length under the old weights, with an additive correction based on the path's endpoints. This correction is the same for all paths from s (= v_1) to t (= v_k). So the shortest path from s to t is the same path under both w and $\widetilde{w}$. Furthermore, there is a negative cycle under w if and only if there is a negative cycle under $\widetilde{w}$ (since $v_k = v_1$ in a cycle, so $\phi(v_k) - \phi(v_1) = 0$).

A^* [Hart, Nilsson, Raphael, 1968]

The A^* algorithm can be seen as a tuning method for Dijkstra for the (often occurring) case where one is searching for a path from s to a specific target node t.

- New ingredient: For all nodes v, attempt to make a guess $h(v)$ at the shortest distance from v to t, i.e., a guess at $\delta(v, t)$. $h(v)$ is also called a heuristic.
- Intuition: If $v.d$ (as in Dijkstra, when v is extracted from PQ) is equal to $\delta(s, v)$, then $v.d + h(v)$ is a guess at $\delta(s, v) + \delta(v, t)$, which is the length of the shortest path from s to t through v.
- Idea: Proceed as in Dijkstra (including the same update of $v.d$-values), but let the key in PQ be $v.d + h(v)$. That is, expand the search via nodes that are guessed to be on the shortest path from s to t. For comparison, Dijkstra can be said to expand via nodes that are known to be the closest to s.

  Example:

  Consider a grid-based graph. Nodes = the white grid cells, edges with length one between white neighboring cells. The heuristic $h(v)$ is equal to the Euclidean distance (straight-line distance) from cell v to the target cell t.

  - Dijkstra (right figure): Investigates uniformly in all directions.
  - *$A^$ with the above heuristic (left figure):** Investigates more towards the goal. Fewer nodes are visited, therefore faster in practice.

Correctness and Worst-Case Runtime of A^*

A heuristic is called consistent if for all nodes v and all edges (v, u) in v's neighbor list:

$$h(v) \leq w(v, u) + h(u)$$

That is, the heuristic's guess (at the shortest path to t) for v's neighbors is not in contradiction with the heuristic's guess (at the shortest path to t) for v.

It can be shown that with a consistent heuristic, A^* is the same as Dijkstra on a graph with adjusted weights. From this, one can show correctness (that the shortest path between s and t is returned by A^*) and that the worst-case runtime is equal to the worst-case runtime for Dijkstra.