Algorithms on Graphs: Part III

Shortest Path Problems

Path in a graph. Let $G = \langle V, E\rangle$ be a graph. A path $p = e_1, \ldots, e_k$, $e_i \in E$, $e_i = (u_i, v_i)$ is a sequence of edges, such that $v_1 = u_2, v_2 = u_3, \ldots, v_{k-1} = u_k$.

Shortest Path. Consider a directed edge-weighted graph $G = \langle V, E, w\rangle$. Let $p = e_1, \ldots, e_k$ be a *path* in G. The weight of the path,

$$w(p) = w(e_1) + \ldots + w(e_k).$$

The **shortest path weight** for a pair (u, v) of edges is defined as:

$$\delta(u, v) = \infty, \text{ if there is no path from } u \text{ to } v;$$
$$\delta(u, v) = \min(w(p)): \text{ p connects } u \text{ to } v.$$

A **shortest path** between u and v is any path p between them, such that $w(p) = \delta(u, v)$.

Variants of the Shortest Path Problem.

Single-source shortest path problem. Given a graph G and a vertex $v \in V$, *find the shortest path from v to every other node in the graph.*

Single-destination shortest path problem. Given a graph G and a vertex $v \in V$, *find the shortest path to v from every other node in the graph.*

Single-pair shortest path problem. Given a graph G and a pair u and v of nodes, *find the shorest path between them.*

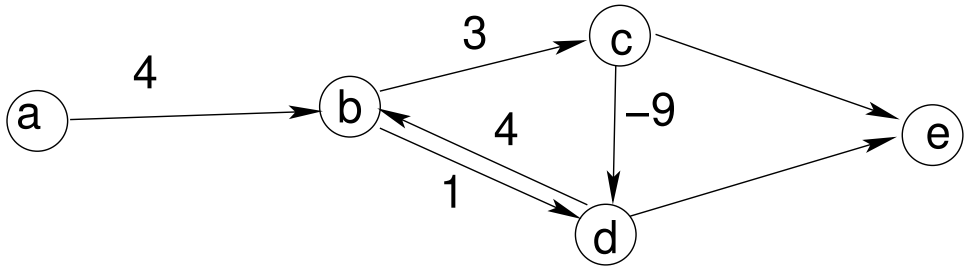

Figure 1: Effect of negative cycles in the graph on shortest paths.

All-pairs shortest paths problem. Given a graph G, find a shorest path *for and pair* u, v *of nodes in* V.

Single-source Shortest Path Problem

Single-source Shorthest Path problem provides an immediate solution to the **single-destination shortest path problem**.

Additionally, no known algorithm for **single-pair shortest path problem** can perform with a better worst-case complexity than the single-source shortest path problem.

Negative Cycles. Shortest paths in a graph **do not exist** if a graph contains a negative-weight cycle.

Figure **??** shows an example of a graph with a negative cycle.

Here, the $(b, c), (c, d), (d, b)$ cycle has a negative weight of -2.

This means that the weight of the $(a, b), (b, c), (c, e)$ path from a to e is going to be smaller than the weight of the $(a, b), (b, c), (c, d), (d, b), (b, c), (c, e)$. This can be continued.

Negative edge weights are OK if they do not lead to a negative cycle.

Bellman-Ford Algorithm. Solves single-source shortest path problem for **any** input (edges can have negative weights).

Dijkstra's Algorithm. Solves single-source shortest path problem for graphs with no negative weights.

Preliminaries

Cycle Elimination. Shortest paths need not contain cycles.

- Negative edge cycles. Negative edge cycles lead to no shortest paths.
- Positive edge cycles. These cycles *add non-trivial value* to the path weight. When excluded, the weight of the path decreases.
- 0-weight edge cycles. These can be excluded from the paths without changing the weight of the path.

Path representation. We represent single-source shortest paths using two arrays $d[V]$ and $\pi[V]$.

$d[v]$ stores the value of the currently known shortest path weight. for the paths from the source node s to node v.

$\pi[v]$ stores the "parent" of v in the shortest path from s to v.

Initialization. All single-source shortest path algorithms will be initialized in the same way:

- $d[v]$ is set to ∞ for all $v \neq s$; $d[s]$ is set to 0.
- $\pi[v]$ is set to NIL for all nodes.

The pseudocode of this procedure is:

```
Procedure INITIALIZE_SINGLE_SOURCE(V, E, w, s)
begin
  foreach v ∈ V do
    d[v] ← ∞;
    π[v] ← NIL;
  endfor
  d[s] ← 0;
end
```

Relaxation. Consider some state of the single-source shortest path computation, defined by a pair of arrays $\pi[V]$ and $d[V]$. Consider some edge (u, v), that has not been considered yet.

Two cases are possible:

- $d[u] + w(u, v) < d[v]$.

  In this case, *the currently known best path from s to u followed by (u, v) is shorter, than the previously discovered shortest path from s to* v. The d and π arrays need to be updated.

- $[u] + w(u, v) \geq d[v]$.

  In this case, the best known path from s to u followed by (u, v) is **not better** than the best known path from s to v that avoids (u, v). No changes to the arrays is necessary.

The **relaxation** step is the procedure that, given an edge in the graph, checks the two conditions above, and if the first condition is true, updates the d and π arrays.

The pseudocode is shown below.

```
Procedure RELAX(u, v, w)
begin
  if d[u] + w(u, v) < d[v] then
    d[v] ← d[u] + w(u, v);
    π[v] ← u;
  endif
end
```

Properties of relaxation. These properties will allow us to formally prove correctness of our single-source shortest path algorithms.

Triangle inequality: For any edge $(u, v) \in E$,

$$\delta(s, v) \leq \delta(s, u) + w(u, v).$$

Upper-bound property. It is always the case that

$$v[d] \geq \delta(s, v)$$

No-path property: If s and v are not connected then

$$\delta(s, v) = d[v] = \infty.$$

Convergence property: If $s \Rightarrow u \rightarrow v$ is the shortest path from s to v, *and* if $d[u] = \delta(s, u)$ at some point in the computation, than after *the relaxation* $\mathsf{RELAX}(u, v, w)$, $d[v] = \delta(s, v)$ from then on.

Path-relaxation property: If $p = s, v_1, \ldots, v_k$ is the shortest path from s to v_k, *and* the edges of p are *relaxed in order* $(s, v_1), (v_1, v_2), \ldots (v_{k-1}, v_k)$, then $d[v_k] = \delta(s, v)$ after all these relaxations.

Predecessor-subgraph property: Once $d[v] = \delta(s, v)$ for all $v \in V$, $\pi[V]$ is the predecessor subgraph for all shortest paths from s to all other nodes.

(Note: these properties require prrof, before we can use them to prove correctness.)

The Bellman-Ford Algorithm

Applicability. The Bellman-Ford Algorithm is applicable to all directed edge-weighted graphs. If a graph has a *negative cycle*, then the algroithm will fail and exit. Otherwise, the algorithm will compute the shortest distance from the source to every node in the graph.

Idea. Exploit the **path relaxation property**. We would love to know the order of vertices on every shortest path from s to other nodes v, so that we coud relax the appropriate edges in a row. This, however is not possible right away. But if we relax every single edge *multiple times*, then the "correct" order of vertices will be present across multiple passes.

Pseudocode. The input to the algorithm Bellman_Ford as shown below:

- $G = \langle V, E, w \rangle$ is the input edge-weighted graph.
- s is the source node.

```
Algorithm Bellman_Ford(V, E, w, s)
begin
  d[1..|V|];
  π[1..|V|];
  INITIALIZE_SINGLE_SOURCE(V,E,w,s);
  for i ← 1 to |V| − 1 do
    foreach (u, v) ∈ E do
      RELAX(u,v,w);
    endfor;
  endfor
  foreach (u, v) ∈ E do
    if d[v] > d[u] + w(u, v) then    //Check for negative weight cycle
      return FALSE;
    endif
  endfor
  return TRUE
  // Array d[.]  will contain shortest distances
  // Array π[.]  will contain shortest shortest path info

end
```

Correctness. Correctness is proven as follows.

Lemma. Let $G = \langle V, E, w \rangle$ be an edge-weighted directed graph, and let $s \in V$ be a source node. Then, *if G has no negative-weight cycles*, then after $|V| - 1$ iterations of the nested loop of Bellman_Ford,

for each node $v \in V$ reachable from s, $d[v] = \delta(s, v)$.

Proof. Let $v \in V$ be reachable from s, and let $p = s, v_1, \ldots, v_{k-1}, v$ be some *shortest path* from s to v.

Shortest paths are simple (recall that we eliminated al cycles from them). Therefore p has at most $|V| - 1$ edges.

On iteration i of Bellman_Ford algorithm we relax every single edge, include the edge (v_{i-1}, v_i) from the path p.

Then, by path relaxation property, $d[v] = \delta(s, v)$.

Complexity. The running time of the Bellmain_Ford algorithm is dominated by the double nested loop. The outer loop repeats $|V| - 1$, i.e. $\Theta(|V|)$ times. On each outer loop iteration, the inner loop goes through all the edges, i.e., repeats $\Theta(|E|)$ times. Thus,

$$T(|V|, |E|) = O(|V||E|)$$

Dijkstra's Algorithm

Applicability. This algorithm only works on graphs $G = \langle V, E, w \rangle$ where $(\forall e \in E) w(e) \geq 0$.

Idea. If there are no need to worry about negative weight cycles (or, in fact, about negative edges), then we do not have to examine each edge on each iteration. If we start at the source node, we are guaranteed that one of its outgoing links (the shortest) will be a shortest path to some other node. This gives us an idea for a **greedy approach**: on each step, maintain

- The set S of nodes for which shortest paths have already been established.
- The shortest paths to all other nodes *only through nodes in* S.

Then, on each step, we pick the node with shortest path through nodes in S, add it to S. Once it is in S, its outgoing edges may become parts of the shortest paths from s to other nodes (not yet in S), so we relax each outgoing edge.

Pseudocode. This algorithm takes the following inputs.

The input to the algorithm DIJKSTRA as shown below:

- $G = \langle V, E, w \rangle$ is the input edge-weighted graph.
- s is the source node.

```
Algorithm DIJKSTRA(V, E, w, s)
begin
  d[1..|V|];
  π[1..|V|];
  INITIALIZE_SINGLE_SOURCE(V,E,w,s);
  S ← ∅;      //Initialize Set of Completed Nodes
  Queue ← V;     //Initialize Priority Queue
  while Queue ≠ ∅ do
    u ← EXTRACT_MIN(Queue);
    S ← S ∪ {u};
    foreach (u, v) ∈ E do
      RELAX(u,v,w);
    endfor
  endwhile
  // Array d[.]  will contain shortest distances
  // Array π[.]  will contain shortest shortest path info
end
```

The algorithm maintains a *priority queue* $Queue$ to store the set of for which shortest paths have not yet been discovered and a set S of nodes for which the shortest paths *have* been discovered. $Queue$ uses the values of $d[.]$ as the priority keys, and retrieves the nodes with the smallest value of $d[.]$. RELAX(...) adjusts the values of $d[.]$. This means that a number of DECREASE_KEY operations implicitly happen in $Queue$.

Correctness. Shown as follows.

Theorem. Dijkstra's algorithm, run on a directed, edge-weighted graph $G = \langle V, E, w \rangle$ with non-negative weight function w and source s terminates with $d[v] = \delta(s, v)$ for all $v \in V$.

Proof. We prove the theorem by showing that our greedy choice property works:

at the start of each **while** loop iteration $d[v] = \delta(s, v)$ for all $v \in S$.

Step 1. Iteration 0: $S = \emptyset$, therefore our loop invariant is true.

Step 2. Iteration $k > 0$. Proof by contradiction. Assume that on some iteration $k > 0$, a node u is added to S, but $d[u] > \delta(s, u)$. Let u be *the first such node.*

Observation 1: $u \neq s$. s is extracted from *Queue* on iteration 1, because at this point $d[v] = \infty$ for all nodes but s, and $d[s] = 0$. It is also true that $d[s] = \delta(s, s)$. Since $u \neq s$, u has been retrieved on an iteration $k > 1$.

Observation 2: When u is added to S, $S \neq \emptyset$, and therefore, $v[u] \neq \infty$, i.e., *there exists a path from s to u.* (if this is NOT the case, $v[u] = \infty = \delta(s, u)$, as the shortest path from s to an unreachable node is ∞).

Consider the shortest path p from s to u and consider the first time p crosses from S to $V - S$. Let y be the first node in $V - S$, p crosses into and let the incoming edge on the path be (x, y) $(x \in S)$.

The path p can be decomposed as $p : s \Rightarrow x \rightarrow y \Rightarrow u$.

Observation 3: When u is added to S, $d[y] = \delta(s, y)$. Because $x \in S$, we know (by induction/loop invariant) that $d[x] = \delta(s, x)$. Because (x, y) is relaxed when x is added to S, but *convergence property*, $\delta(s, y) = d[y]$.

Observation 4: Contradiction:

$$d[y] = \delta(s, y) \leq \delta(s, u) \leq d[u].$$

BUT, u is chosen on step k from the priority queue, which implies that $d[u] \leq d[y]$. From here, it must be the case that

$$d[y] = \delta(s, y) = \delta(s, u) = d[u].$$

This **contradicts** our assumption that $d[u] \neq \delta(s, u)$ when u is included in S.

Complexity. Dijkstra's algorithm maintains a *priority queue*, so its computational complexity depends on the implementation of the queue. The following priority queue operations are used in the algorithm (implicitly, due to the powers of pseudocode, or explicitly):

- INSERT() is called implicitly $|V|$ times in the *Queue* $\leftarrow V$ line.
- FIND_MIN() is called explicitly *once on each iteration.* There are $|V|$ iterations of the **while** loop.
- DECREASE_KEY() is called implicitly from RELAX() any time and edge needs to be relaxed. This can be as often as $|E|$, i.e., in the worst case, DECREASE_KEY() is called $\Theta(|E|)$ times.

Priority Queue	T(INSERT())	T(FIND_MIN())	T(DECREASE_KEY())	T(DIJKSTRA)
Array	$O(1)$	$O(\|V\|)$	$O(1)$	$O(\|V\|^2 + \|E\|)$
Binary heap	$O(\log_2(\|V\|)$	$O(\log_2(\|V\|)$	$O(\log_2(\|V\|)$	$O(\|V\| \log_2(\|V\|) + \|E\| \log_2(V))$
Fibonacci heap	$O(1)$	$O(\log_2(\|V\|)$	$O(1)$	$O(\|V\| \log_2(\|V\|) + \|E\|)$