5 Shortest Paths

In this chapter, we resume our study of graph algorithms from Ch. 2. In particular, we examine multiple algorithms that find shortest paths. As we've seen before, if all edges have the same length, then BFS is perfectly fine. However, the problem gets more complicated when the edge lengths are different (and possibly negative!).

The algorithms in this chapter only compute the *length* of a shortest path, rather than the path itself. But as we've seen already, we can maintain parent pointers and trace them at the end to recover the shortest paths themselves.

5.1 Paths in a DAG

Problem statement. Let $G = (V, E)$ be a directed acyclic graph (DAG) where each edge $e = (u, v)$ has (possibly negative) length $\ell(u, v) = \ell_e$, and let s be a starting vertex. Our goal is to find the shortest path from s to all other vertices.

Algorithm. We assume that the vertices of G are ordered in a way such that all edges are oriented from "left to right." This is known as a *topological order*; see Sec. 2.4, Exercise 2.17. We can also assume that s is the first vertex in this order, since no vertex can reach any vertex appearing before it in the order. For each vertex v in this order, and each edge $e = (u, v)$ entering v, we "relax" e (more on this later) by setting $d[v] = \min(d[v], d[u] + \ell_e)$.

Algorithm 14 Shortest Paths in a DAG

Input: DAG $G = (V, E)$, edge lengths ℓ, starting vertex s

```
1: Topologically sort the vertices V
2: d[s] ← 0, d[v] ← ∞ for each v ∈ V
3: for each vertex v in V \ {s} do
4:     for each edge (u, v) do
5:         d[v] ← min(d[v], d[u] + ℓ(u, v))        ▷ "relax" the edge (u, v)
6: return d
```

Theorem 5.1. *At the end of Algorithm 14, $d[v]$ contains the distance from s to v.*

Proof. Notice that Algorithm 14 is a dynamic program! Let $\mathsf{OPT}[v]$ denote the distance from s to v. Then $\mathsf{OPT}[s] = 0$ and for $v \neq s$, the recurrence is

$$\mathsf{OPT}[v] = \min_{(u,v)\in E} d[u] + \ell(u, v).^1$$

This is because the shortest path from s to v must "come from" some vertex u (possibly s) before traversing (u, v) to arrive at v. Thus, $\mathsf{OPT}[v]$ corresponds to the best option across all edges entering v. The algorithm computes d according to this recurrence. □

[1] Notice that this recurrence is very similar to the one for LIS from Sec. 4.2.

Edge relaxations. To *relax* a (directed) edge (u, v) means setting $d[v] = \min(d[v], d[u] + \ell(u, v))$. Relaxing is beneficial because if u is the penultimate node on a shortest s-v path and $d[u] = \delta(s, u)$, then relaxing (u, v) results in $d[v] = \delta(s, v)$. This allows us to proceed in a BFS-like manner, starting from s and "moving out." We will see this process again in the next two sections.

5.2 Dijkstra's algorithm

Problem statement. Let $G = (V, E)$ be a directed graph where each edge $e = (u, v)$ has length $\ell(u, v) \geq 0$, and let s be a starting vertex. Our goal is to find the shortest path from s to all other vertices.

Algorithm. We present Dijkstra's algorithm; notice its similarities to both BFS (Sec. 2.1) and Prim's algorithm (Sec. 2.3). It starts by adding s to a set S, and in each step, adds the vertex $u \in V \setminus S$ with the smallest d value. It then relaxes the edges leaving u and repeats.

Algorithm 15 Dijkstra's Algorithm

Input: Graph $G = (V, E)$, nonnegative edge lengths ℓ, vertex $s \in V$

1: $d[s] \leftarrow 0$, $d[v] \leftarrow \infty$ for each $v \in V \setminus \{s\}$
2: $S \leftarrow \emptyset$, $Q \leftarrow V$
3: **while** $|Q| \geq 1$ **do** ▷ Q contains the unprocessed vertices
4: $u \leftarrow \arg\min_{v \in Q} d[v]$
5: Remove u from Q, add u to S ▷ S contains the processed vertices
6: **for each** out-neighbor v of u **do**
7: $d[v] \leftarrow \min(d[v], d[u] + \ell(u, v))$ ▷ "relax" the edge (u, v)
8: **return** d

Theorem 5.2. *At the end of Dijkstra's algorithm, for every vertex u, $d[u]$ contains the distance from the starting vertex s to u.*

Proof. Let $\delta(u)$ denote the distance from s to u; we will show that when u gets added to S, $d[u] = \delta(u)$. To start, the first vertex added to S is s, and we manually set $d[s] = \delta(s) = 0$ in Line 1. Now suppose u is the $(k+1)$-th vertex added to S. Let P denote the s-u path found by the algorithm, and let P^* denote any s-u path. Since u is not in S yet, P^* must leave S through some edge (x, y); let P^*_x denote the s-x subpath of P^*. Inductively,

$$d[x] + \ell(x, y) = \delta(x) + \ell(x, y) \leq \ell(P^*_x) + \ell(x, y) \leq \ell(P^*).$$

When x was added to S, we relaxed the edge (x, y), so $d[y] \leq d[x] + \ell(x, y)$. And in the current iteration, the algorithm is removing u from Q (rather than y), so $d[u] \leq d[y]$. Putting this all together, we have $d[u] \leq \ell(P^*)$, so P is indeed a shortest path and $d[u] = \delta(u)$. □

5.3 Bellman-Ford

Problem statement. Let $G = (V, E)$ be a directed graph where each edge $e = (u, v)$ has (possibly negative) length $\ell(u, v)$, and let s be a starting vertex. Our goal is to find the shortest path from s to all other vertices.

Remark. We assume that G has no "negative cycle" (i.e., a cycle whose total weight is negative) since the existence of such a cycle muddles the meaning of "shortest path": one "path" from s to t could traverse a negative cycle an arbitrary number of times, making its length extremely negative. In this case, it's not even clear what "shortest path" means.[2]

Algorithm. The algorithm is quite simple: do $n-1$ iterations, and in each iteration, relax every edge of the graph. The total running time is $O(mn)$.

Algorithm 16 Bellman-Ford

Input: Graph $G = (V, E)$, edge lengths ℓ, vertex $s \in V$

1: $d[s] \leftarrow 0$, $d[v] \leftarrow \infty$ for each $v \in V \setminus \{s\}$
2: **for** $n - 1$ iterations **do**
3: **for each** edge $(u, v) \in E$ **do**
4: $d[v] \leftarrow \min(d[v], d[u] + \ell(u, v))$ ▷ "relax" the edge (u, v)
5: **return** d

Before formally proving the theorem, let's intuitively analyze the algorithm. Consider a shortest path $P = (s, u_1, u_2, \ldots, u_k)$ from s to $v = u_k$. In the i-th iteration, the edge (u_{i-1}, u_i) is relaxed, so $d[u_i]$ is set to $\delta(u_i)$, the distance from s to u_i. Since $k \leq n - 1$, after $n - 1$ iterations, every vertex $u_i \in P$ satisfies $d[u_i] = \delta(u_i)$. In particular, $d[v] = \delta(v)$.

Theorem 5.3. *At the end of the Bellman-Ford algorithm, for every vertex v, $d[v]$ is the distance from the starting vertex s to v.*

Proof. Let $\lambda(i, v)$ denote the length of the shortest *s-v walk* (i.e., path that allows repeated vertices) that contains at most i edges. (If no such walk exists, $\lambda(i, v) = \infty$.) Since G has no negative cycle, a shortest s-v walk is also a shortest s-v path, so it has at most $n - 1$ edges. Thus, it suffices to prove that after i iterations, $d[v] \leq \lambda(i, v)$; the theorem follows from $i = n - 1$. We proceed by induction.

If $i = 0$, then $d[s] = 0 \leq 0 = \lambda(i, s)$ and $\lambda(i, v) = \infty$ for all $v \in V \setminus \{s\}$, so $d[v] \leq \lambda(0, v)$. Now consider $i \geq 1$, a vertex v, and a shortest s-v walk W containing at most i edges. If W does not exist, then $d[v] \leq \infty = \lambda(i, v)$. Otherwise, let (u, v) be the last edge of W, so the length of W satisfies the recurrence

$$\lambda(i, v) = \lambda(i - 1, u) + \ell(u, v).$$

Inductively, we have $d[u] \leq \lambda(i - 1, u)$. So in the i-th iteration, when relaxing (u, v), we set $d[v] \leq \lambda(i - 1, u) + \ell(u, v) = \lambda(i, v)$, as desired. □

5.4 Floyd-Warshall

Problem statement. Let $G = (V, E)$ be a directed graph where each edge $e = (u, v)$ has (possibly negative) length $\ell(u, v)$. Our goal is to find the shortest path between every pair of vertices. (As discussed in the previous section, we assume that G has no negative cycle.)

[2] Actually, a path is not allowed to repeat any vertex, so the shortest path is technically still well-defined. What we really mean is that there is no shortest *walk*, but overloading the term "path" this way is very common. Finding the actual shortest *path* in polynomial time is likely impossible when a negative cycle could exist (see Ch. 8).

Remark. One solution for this problem is to run the Bellman-Ford algorithm (or Dijkstra's, if all lengths are nonnegative) from every vertex. However, this approach can be very slow if m is large; in this case, the Floyd-Warshall algorithm is faster since its running time does not depend on m.

Algorithm. We'll use dynamic programming again. Suppose each vertex has a distinct label in $[n] = \{1, 2, \ldots, n\}$. Let $\delta^r(u, v)$ denote the length of the shortest u-v path that only uses a subset of $\{1, 2, \ldots, r\}$ as intermediate vertices, so $\delta(u, v) = \delta^n(u, v)$. The base case is that $\delta^0(u, v) = \ell(u, v)$ if (u, v) is an edge and ∞ otherwise. For $r \geq 1$, the path corresponding to $\delta^r(u, v)$ either ignores vertex r or passes through it. Thus, we have the following recurrence:

$$\delta^r(u, v) = \min\left(\delta^{r-1}(u, v), \delta^{r-1}(u, r) + \delta^{r-1}(r, v)\right).^3$$

We can use this recurrence to compute all $\delta^r(u, v)$ in $O(n^3)$ time, and as a reminder, we can maintain pointers to recover the shortest paths themselves.

Algorithm 17 Floyd-Warshall

Input: Graph $G = (V, E)$, edge lengths ℓ

Relabel V so that $V = \{1, \ldots, n\}$
Initialize $d^r[u, v] \leftarrow 0$ for each $r \in \{0, 1, \ldots, n\}$ and $u, v \in [n]$
for $u \leftarrow 1, \ldots, n$ **do**
    for $v \leftarrow 1, \ldots, n$ **do**
        if $(u, v) \in E$ **then**
            $d^0[u, v] \leftarrow \ell(u, v)$
        else
            $d^0[u, v] \leftarrow \infty$
for $r \leftarrow 1, \ldots, n$ **do**
    for $u \leftarrow 1, \ldots, n$ **do**
        for $v \leftarrow 1, \ldots, n$ **do**
            $d^r[u, v] \leftarrow \min(d^{r-1}[u, v], d^{r-1}[u, r] + d^{r-1}[r, v])$
return d^n

Remark. Our approach uses $n + 1$ three-dimensional tables indexed by $r \in \{0, 1, \ldots, n\}$. We can alternatively think of d as a single, three-dimensional table indexed by u, v, r, so the recurrence would be

$$d[u, v, r] = \min\left(d[u, v, r-1], d[u, r, r-1] + d[r, v, r-1]\right).$$

[3] Notice that this recurrence is similar to that of the Knapsack DP.

5.5 Exercises

5.1. Consider the same setup as Sec. 5.1, but now our goal is to find the *longest* path from s to all other vertices. Give a linear-time algorithm for this problem.

5.2. Consider the following alternative to Bellman-Ford: add a large constant to every edge lengths so that all edge lengths are nonnegative, and then run Dijkstra's algorithm. Show that this algorithm does not necessarily return a correct solution.

5.3. Let $G = (V, E)$ be a directed graph where each edge $e = (u, v)$ has an *integer* length $\ell(u, v) \leq L$ where L is a constant, and let s be a starting vertex. Show how BFS can be used to find the shortest path from s to all other vertices in $O(m)$ time.

5.4. Notice that Algorithm 14 looks at edges entering each vertex, while Algorithms 15 and 16 look at edges leaving each vertex. Redesign Algorithm 14 so that it also looks at edges leaving each vertex, and prove that the resulting algorithm is still correct.

5.5. Show that Dijkstra's algorithm can be implemented to run in $O(n^2)$ time (where n is the number of vertices in the graph, as usual).

5.6. Give an $O(mn)$-time dynamic programming version of the Bellman-Ford algorithm.

5.7. Let G be a directed graph with nonnegative edge lengths, and let x be a vertex of G. An *x-walk* is a walk that passes through x. Give an $O(n^2)$-time algorithm that finds the length of the shortest x-walk between every pair of vertices.

5.8. Let G be a directed graph with nonnegative edge lengths, and let s be a vertex of G. Give an algorithm that finds the shortest cycle containing s. The running time should be asymptotically the same as that of Dijkstra's algorithm.

5.9. Let G be a directed graph with nonnegative edge lengths, and let s, t be vertices of G. Give an algorithm that finds the shortest walk containing an even number of edges between s and t (or reports that no such walk exists). The running time should be asymptotically the same as that of Dijkstra's algorithm.

5.10. Consider the Bellman-Ford algorithm, but we run it for one extra iteration at the end. Prove that the graph has a negative cycle reachable from s if and only if some d-value gets updated in the n-th iteration. Use this to design an $O(mn)$-time algorithm that detects whether or not a graph has a negative cycle.

5.11. Suppose you have already ran the Floyd-Warshall algorithm to obtain a matrix d where $d[u, v] = \delta(u, v)$ for every $u, v \in V$. Now suppose the length of an edge e decreases from $\ell(e)$ to $\ell'(e)$. Give an $O(n^2)$-time algorithm to update the matrix d for this new graph. (Bonus: what if all lengths are positive, and the graph is undirected?)