4 Paths in Graphs

In this chapter, we resume our study of graph algorithms from Ch. 1. In particular, we examine multiple solutions to the problem of finding the shortest path between two vertices. If all edges have the same length, then BFS solves this problem in linear time (see Sec. 1.1). However, things get more complicated when the edge lengths differ.

In Sec. 4.1, we solve the problem for directed acyclic graphs. In Sec. 4.2, we solve the problem for general graphs and nonnegative edge lengths. In Sec. 4.3, we remove the assumption that edge lengths are nonnegative. Finally, in Sec. 4.4, we show how we can simultaneously calculate the shortest path between all pairs of vertices.

Throughout this chapter, we let $\delta(s,t)$ denote the length of the shortest path from s to t. Furthermore, the algorithms we study only compute the *length* of a shortest path, rather than the path itself. But as we saw in Ch. 3, we can maintain parent pointers and trace them at the end to recover the shortest paths themselves.

4.1 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 length of the shortest path from s to all other vertices.

Algorithm. First, topologically sort the vertices of G such that no edges enter the first vertex (see Sec. 1.4 Exercise 4).[1] Then, 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 8 Shortest Paths in a DAG

```
1: Use DFS to topologically sort the vertices V.
2: Initialize d[v] = ∞ for all v ∈ V.
3: for each vertex v in topological order do
4:     if v = s then
5:         Set d[v] = 0.
6:     else
7:         for each edge (u, v) do
8:             Set d[v] = min(d[v], d[u] + ℓ(u, v)).        ▷ "relax" the edge (u, v)
9: return d
```

Theorem 4.1. *At the end of Algorithm 8, $d[v]$ contains the distance from s to v.*

Proof. Notice that Algorithm 8 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

$$\mathsf{OPT}[v] = \min_{(u,v)\in E} d(u) + \ell(u, v).^2$$

[1] Note that we might as well assume this first vertex is s, because no vertices that appear before s can be reached from s.

[2] Notice that this recurrence is very similar to the one for LIS, from Sec. 3.1.

This is because the shortest path from s to v must go to some intermediate vertex u (possibly s if $(s, v) \in E$) and then traverse the edge (u, v). Thus, $\mathsf{OPT}[v]$ is simply the best option across all possible edges entering v. Since the algorithm calculates d according to this recurrence, we've proven the theorem. □

Edge relaxations. To *relax* a (directed) edge (u, v) means setting $d[v] = \min(d[v], d[u] + \ell(u, v))$. This is a useful procedure: if u is the penultimate node on a shortest s-v path and $d[u] = \delta(s, u)$, then relaxing (u, v) results in $d[v] = \delta(s, v)$. We will see this process being used again in both Dijkstra's algorithm and the Bellman-Ford algorithm.

4.2 Dijkstra's algorithm

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

Algorithm. We present Dijkstra's algorithm; notice its similarities to both BFS (Sec. 1.1) and Prim's algorithm (Sec. 1.3). It starts by adding s to a set S, and in each step, adds a vertex in $V \setminus S$ whose distance from s through vertices in S is minimized.

Algorithm 9 Dijkstra's Algorithm

```
Initialize d[s] = 0 and d[v] = ∞ for all v ∈ V \ {s}.
Initialize sets S = ∅ and Q = V.
while |Q| ≥ 1 do                          ▷ Q contains the unprocessed vertices
    Let u = arg min_{x∈Q} d[x].
    Remove u from Q and add u to S.       ▷ S contains the processed vertices
    for each neighbor v of u do
        Set d[v] = min(d[v], d[u] + ℓ(u, v)).      ▷ "relax" the edge (u, v)
return d
```

Remark. As mentioned earlier, whenever we update the value $d[v]$, we can direct a pointer from v to its parent u that is being processed in this iteration of the while loop. By following these pointers back to s, we can recover the shortest path from s to every vertex in V.

Theorem 4.2. *At the end of Dijkstra's algorithm, for every vertex u, $d[u]$ contains the distance from the starting vertex s to u.*

Proof. We will show, by induction on $|S|$, that whenever a vertex u gets added to S, we have $d[u] = \delta(u)$ where $\delta(u)$ denotes the distance from s to u. When $|S| = 1$, then s is the only vertex in S, and we indeed have $d[s] = \delta(s) = 0$.

Now suppose $|S| = k$ and u is the $(k + 1)$-th vertex that we are adding to S. Let P denote the s-u path that the algorithm has found, 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 in P^*. By the inductive hypothesis, we have

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

When x was added to S, we relaxed the edge (x, y), so $d[y] \leq d[x] + \ell(x, y)$. Finally, in this 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. □

4.3 The Bellman-Ford algorithm

Problem statement. Let $G = (V, E)$ be a directed graph 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 length of the shortest path from s to all other vertices.

Remark. We assume that G has no cycle whose total weight is negative, because the existence of such a cycle muddles the meaning of "shortest path." To go from s to t, we could first go to the negative cycle and traverse it any arbitrarily large number of times. Then the resulting "path" has very negative cost, so it's not even clear what to return.[3]

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

Algorithm 10 Bellman-Ford

```
Initialize d[s] = 0 and d[v] = ∞ for all v ∈ V \ {s}.
for n − 1 iterations do
    for each edge (u, v) ∈ E do
        Set d[v] = min(d[v], d[u] + ℓ(u, v)).        ▷ "relax" the edge (u, v)
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 $t = u_k$. In the i-th iteration, the edge (u_{i-1}, u_i) is relaxed and $d[u_i]$ is set to $\delta(s, u_i)$. Thus, after $k \leq n - 1$ iterations, every vertex u_i on P satisfies $d(u_i) = \delta(s, u_i)$. In particular, $d(t) = \delta(s, t)$.

Theorem 4.3. *At the end of the Bellman-Ford algorithm, for every vertex v, $d[v]$ contains the distance from the starting vertex s to v.*

Proof. Let $\lambda(i, v)$ denote the length of the shortest *walk* (i.e., sequence of adjacent vertices with repetition allowed) from s to v that contains at most i edges. Since the shortest walk from s to v contains at most $n - 1$ edges, it is sufficient to prove that after i iterations, $d[v] \leq \lambda(i, v)$. 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

[3] Actually, vertices are not allowed to repeat in a path, 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* is very difficult when the graph has negatively weighted edges.

$d[v] \leq \lambda(0, v)$. Now consider $i \geq 1$, a vertex v, and a shortest walk W from s to v containing at most i edges. If the last edge of W is (u, v), then

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

By the inductive hypothesis, 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. □

4.4 The Floyd-Warshall algorithm

Problem statement. Let $G = (V, E)$ be a directed graph where each edge $e = (u, v)$ has (possibly negative) length $\ell(u, v) = \ell_e$. Our goal is to find the length of the shortest path between every pair of vertices. (As discussed in Sec. 4.3, we assume that the graph has no negative cycles.)

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; the following algorithm is both simple and faster when the graph has many edges.

Algorithm. Once again, we'll use dynamic programming. Suppose the vertices are numbered $1, 2, \ldots, n$. Let $\delta(u, v, r)$ denote the length of the shortest u-v path that only passes through vertices $\{1, 2, \ldots, r\}$, so $\delta(u, v) = \delta(u, v, n)$. Notice that $\delta(u, v, 0)$ is $\ell(u, v)$ if (u, v) is an edge, or ∞ otherwise. For $r \geq 1$, the path corresponding to $\delta(u, v, r)$ either passes through vertex r or it doesn't. Thus, we have the following recurrence:

$$\delta(u, v, r) = \min(\delta(u, v, r-1), \delta(u, r, r-1) + \delta(r, v, r-1)).$$

The Floyd-Warshall algorithm implements this recurrence in $O(n^3)$ time, and as usual, we can maintain pointers and follow them at the end recover the actual shortest paths.

Algorithm 11 Floyd-Warshall

```
1: Number the vertices so that V = {1, ..., n}.
2: for u = 1, ..., n do
3:     for v = 1, ..., n do
4:         Set d[u, v, 0] = ℓ(u, v) if (u, v) ∈ E and ∞ otherwise.
5: for r = 1, ..., n do
6:     for u = 1, ..., n do
7:         for v = 1, ..., n do
8:             Set d[u, v, r] = min(d[u, v, r − 1], d[u, r, r − 1] + d[r, v, r − 1]).
9: return d
```

4.5 Exercises

1. Let $G = (V, E)$ be a directed graph where each edge $e = (u, v)$ has an *integer* length $\ell(u, v) \geq 1$, and let s be a starting vertex. Show how BFS can be used to find the shortest path from s to all other vertices.

2. Consider the same setup as Sec. 4.1, but now our goal is to find the *longest* path from s to all other vertices. Give a linear-time algorithm for this problem.

3. Notice that Algorithm 8 looks at edges entering each vertex, while Algorithms 9 and 10 look at edges leaving each vertex. Redesign Algorithm 8 so that it also looks at edges leaving each vertex, and prove that the resulting algorithm is still correct.

4. 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. 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.

6. 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 shortest x-walk between every pair of vertices.

7. 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 running Dijkstra's algorithm once.

8. 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 running Dijkstra's algorithm once.

9. Give an $O(mn)$-time dynamic programming version of the Bellman-Ford algorithm (Algorithm 10), and prove that it is correct.

10. Consider the Bellman-Ford algorithm, but we run it for one extra iteration at the end. Prove that 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.

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 w_e to w'_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?)