Lecture Notes for CSCI 311: Algorithms

Set 18-Shortest Paths

Professor Talmage

April 23, 2025

---

1 Problem Definitions

There are actually multiple versions of the shortest path problem, of which we will consider two. We return to weighted, directed graphs to consider shortest paths, though our discussion generalizes immediately to undirected graphs, as we can simply consider each undirected edge (u, v) to be the pair of directed edges $(u, v), (v, u)$.

1. *Single-Source Shortest Paths (SSSP)*: From a given start node, what is the minimum distance to every other node in the graph?

2. *All-Pairs Shortest Paths (APSP)*: For every ordered pair of nodes in a graph, what is the minimum distance from the first node to the second?

Neither of these is, perhaps, what you would initially think of. The most intuitive version of the problem is probably Single-Pair Shortest Path, finding the minimum distance from one given start node to a given end node. Note that solving SSSP is sufficient to solve that problem, since we can then simply look up the shortest path for the desired end node. Further, while solving all of SSSP may seem excessive, we do not actually know a solution for SPSP that is any more efficient, since the shortest path to our desired destination may go through any other vertices and since the problem has optimal substructure, as we will prove momentarily, we will inherently find the shortest paths to all of those intermediate nodes. (Of course, it is also sufficient to solve APSP, but that tends to be more expensive.)

Due to lack of time, and an abundance of interesting topics, we will focus primarily on SSSP and not spend much time on APSP. Note that, given a good SSSP algorithm, the additional cost to solve APSP is a factor of $|V|$, since we can run SSSP from every possible starting node.

Lemma 1. *Given a weighted digraph $G = (V, E, w : e \to \mathbb{R})$, let $p = (v_0, \ldots, v_k)$ be a shortest path from v_0 to v_k and for all $0 \leq i \leq j \leq k$, let $p_{ij} = (v_i, \ldots, v_j)$ be the subpath from v_i to v_j. Then p_{ij} is a shortest path from v_i to v_j.*

Proof. By contradiction. Assume that for some i, j there is a path p'_{ij} from p_i to p_j that has lower total weight that p_{ij}. Then $p' = p_{0i} \to p'_{ij} \to p_{jk} = (v_0, \ldots, v_i) \to p'_{ij} \to (v_j, \ldots, v_k)$ is a path from v_0 to v_k with lower total weight than p, contradicting our assumption of p's optimality. □

Lemma 2. *If G has no cycles with negative weight, all shortest paths are acyclic and any shortest path has less than or equal to $|V| - 1$ edges.*

Exercise: Prove this claim.

2 Relaxation

Both of the algorithms we will consider use a fundamental technique called *relaxation*. We saw something similar in Prim's algorithm, where we updated distance estimates to each node as we found new ways to reach them from the partial tree we were building, but this is a somewhat more general technique.

We will maintain in each node an estimate of the length of the shortest path from the start node to that node. We *relax* an edge (u, v) by updating our estimate of v's distance to $\min(v.dist, u.dist + w(u, v))$. That is, we check to see if we can get to v more efficiently by going through u as the last stop before v, and if so, reduce our estimate of the distance to v. This is an application of the triangle inequality, which says the direct distance between two points is at most the sum of the other two sides of any triangle containing those points. Mathematically, $dist(s, v) \leq dist(s, u) + w(u, v)$, for any u. Note that this is not the same as the restriction on edge distances we used in recitation for the approximate Traveling Salesman algorithm. Here, we are not adding any restriction on the input graph, merely comparing alternate routes.

3 Algorithms

We will look at two algorithms for SSSP: The *Bellman-Ford* algorithm is a dynamic programming solution and *Dijkstra's* algorithm is a greedy solution (that requires the graph have no negative edge weights).

3.1 Bellman-Ford

Algorithm 1 Bellman-Ford SSSP algorithm.

```
1:  function BELLMAN-FORD(G = (V, E, w), s ∈ V)
2:      for all v ∈ V do                                          ▷ Initialization
3:          v.distance = ∞
4:          v.π = ⊥
5:      end for
6:      s.distance = 0                                            ▷ End initialization
7:      for i = 1 to |V| − 1 do
8:          for all (u, v) ∈ E do
9:              relax(u, v)                  ▷ i.e. v.dist = min(v.distance, u.distance + w(u, v))
10:             if v.distance changed then v.π = u
11:             end if
12:         end for
13:     end for
14:     for all (u, v) ∈ E do
15:         if v.distance > u.distance + w(u, v) then
16:             return False
17:         end if
18:     end for
19:     return True
20: end function
```

Complexity:

Exercise: What is this algorithm's complexity?

We have nested loops, one running on the number of vertices (minus one) and the other running on every edge, then we iterate across the edges one more time, for a total of $O(|V||E|)$.

Correctness: The core idea follows from the lemma you proved above that any shortest path uses at most $|V| - 1$ edges. Thus, in $|V| - 1$ iterations, we have relaxed all of the edges of every path of at most $|V| - 1$ hops, in order. To see this, observe that in the first round of iterations, the only nodes whose distances will update are neighbors of the start node. In the second round, all nodes within two hops will update their distances to the minimum-weight two-hop path length, and so on in later rounds. We then know that after $|V| - 1$ rounds, we have relaxed along every shortest path in order, and thus the distance estimate for every node is in fact the length of the shortest path. The minimum in the relaxation operation means that we never update to a worse path, which makes this a dynamic programming algorithm instead of brute force. Brute-force checking every path of length at most $|V| - 1$ could cost as much as $|V|^{|V|-1}$, which would be much worse.

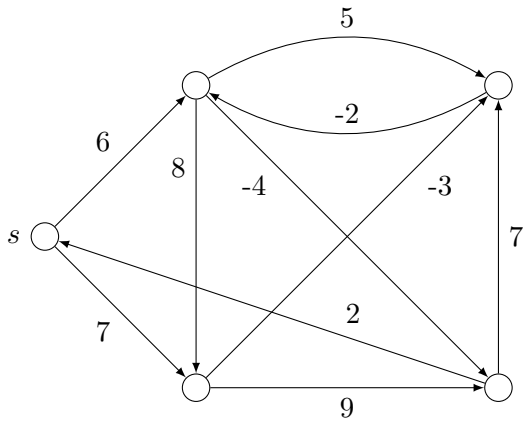
The extra loop at the end relaxing each edge one more time detects negative-weight cycles, as shortest path distances should be fixed at this point. The only way for them to change is if there is a negative-weight cycle, since we can go around that cycle an infinite number of times, decreasing shortest path weight forever.

Exercise: Which representation should we use to store the graph?

We want to be able to iterate across all edges, so we should keep an array E of all edges, but then we need to get $w(u, v)$ quickly, which an adjacency matrix can do well. We simply update it to store $w(u, v)$ at index $[u, v]$, with weight ∞ for non-existent edges.

Example:

Exercise: Run the Bellman-Ford algorithm on the following graph.



3.2 Dijkstra

This is a greedy algorithm similar to Prim's MST algorithm. We explore nodes in order of minimum path length, updating distance estimates to all neighbors of each node as we explore it. This algorithm requires that there be no negative-weight edges in the graph, at all, not just no negative-weight cycles. This guarantees that we cannot later find a shorter path that appeared worse initially, then suddenly got much better.

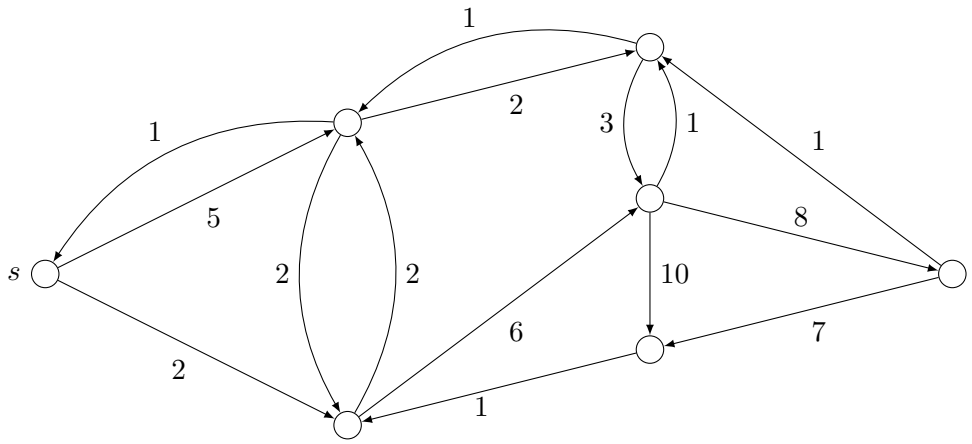
Exercise: Which graph representation will be most efficient?

Since we frequently iterate across the neighbors of a vertex, but never check whether a given edge is in the graph, an adjacency list will be most efficient. We will also need access to a list of vertices to put them in the priority queue, but that is part of an adjacency list representation, so requires no extra effort or space.

Algorithm 2 Dijkstra's SSSP algorithm

```
1: function DIJKSTRA(G = (V, E, w), s ∈ S)
2:     Initialize as for Bellman-Ford
3:     S = ∅
4:     Q = V, min-priority queue keyed on nodes' distance values
5:     while Q is not empty do
6:         u = Q.ExtractMin()
7:         S = S ∪ {u}
8:         for all v ∈ Adj(u) do
9:             Relax(u, v)
10:        end for
11:    end while
12: end function
```

Exercise: Run Dijkstra's algorithm on the following graph.



Complexity: Initialization takes $|V| \log |V|$, since we do constant work per node, then need to insert them all into a priority queue, where each insert takes $O(\log |V|)$ time. The while loop runs $|V|$ times and the for loop runs a total of $|E|$ times, across all vertices. Each *relax* potentially calls $DecreaseKey$ in the priority queue, so we have $O(|V| \log |V|)$ for $ExtractMin$ calls plus $O(|E| \log |V|)$ for relaxation. This gives a total cost of $O((|V| + |E|) \log |V|)$.

Aside: There is another heap implementation, called a Fibonacci Heap, which is optimized for handling many $DecreaseKey$ calls, such as we have here. With that structure, we have amortized cost of $O(1)$ for $DecreaseKey$, so the total cost of Dijkstra's algorithm using a Fibonacci-Heap based priority queue is $O(|V| \log |V| + |E|)$.

Correctness: We can prove correctness using a loop invariant (induction on the iterations), that every node $x \in S$ has $x.distance = \delta(s, x)$, where $\delta(a, b)$ is the actual shortest path length from a to b. That is, when we extract a node from the priority queue, its distance estimate is correct.

- Initialization: S is initially empty, so the invariant holds vacuously.
- Maintenance: The idea is that in each iteration, we add a single node, so we need to show that its distance estimate is correct. Because we take the node with the lowest distance from the priority queue and all edge weights are non-negative, any path through a not-yet resolved node would be at least as long, since the distance to any such node is at least as long as the distance to this node.

- Termination: We stop when all elements are in S, because we have extracted them all from Q.