CS 344: Design and Analysis of Computer Algorithms **Rutgers: Spring 2022**

Lecture 19

March 31, 2022

Instructor: Sepehr Assadi

1 Single-Source Shortest Path

In this lecture, we look into another important concept in graphs, i.e., **shortest path** computation. The problem is simply as follows:

Problem 1. Given a graph $G = (V, E)$ (directed or undirected) with *positive* weight $w_e > 0$ on each edge $e \in E$, and a designated vertex $s \in V$, called a *source*, find the value of the shortest path from s to every vertex $v \in V$, denoted by $dist(s, v)$. In other words, for any path P, we define $w(P) = \sum_{e \in P} w_e$, namely, the total weight of the edges in this path, and our goal is to find the weight of the minimum weight path (minimizing $w(P)$) from s to any other vertex v (thus we are actually looking for minimum weight path but the convention is to still call this shortest path).

Example: The correct answer for the shortest path problem on the following graph is $dist(s, s) = 0$, $dist(s, v_1) = 12$, $dist(s, v_2) = 3$, and $dist(s, v_3) = 8$.

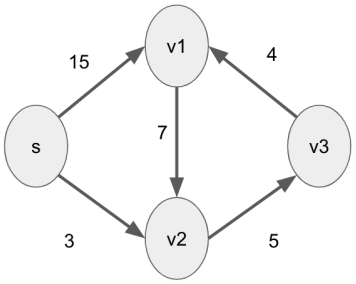


Figure 1: Notice that the shortest path on this weighted graph from s to v_1 is the path $s \to v_2 \to v_3 \to v_1$ with weight 12 and *not* $s \to v_1$ with weight 15.

Shortest path in unweighted graphs? We have already seen a version of the shortest path problem on *unweighted* graphs, which can be solved using the BFS algorithm. We can think of that problem as a special case of Problem 1 where weight of every edge $e \in E$ is $w_e = 1$.

2 Algorithms for Single-Source Shortest Path

In this course, we examine two different algorithms for finding shortest paths, the Bellman-Ford algorithm which is amazingly simple (and has lots of interesting properties for different applications) but is somewhat inefficient and Dijkstra's algorithm which is very similar to Prim's algorithm (and thus is quite simple still) and is also more efficient.

2.1 Bellman-Ford's Algorithm

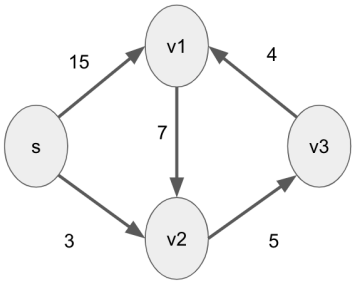
The Bellman-Ford algorithm is based on the following general idea. Suppose we have computed "tentative" shortest path distances from s to every other vertex, namely, $d[v]$ for every $v \in V$. Ideally, we would like to have $d[v] = dist(s, v)$ in which case we will be done. However, we clearly do not know these values at the very beginning of the algorithm. Instead, we are happy by simply having some $d[v]$ for every vertex v *which is equal to weight of some s-v path* in G (but not necessarily the minimum weight one) which means we instead will have $d[v] \geq dist(s, v)$ (and $d[s] = dist(s, s) = 0$). Now in that case, whenever we find an edge $e = (u, v)$ where $d[v] > d[u] + w_e$, we can update $d[v] = d[u] + w_e$: this basically means that we find another path from s to v with smaller weight (by going from s to u and then taking the edge $e = (u, v)$). We can then repeat this process as long as needed, by finding an edge that can update our tentative shortest path distances, and since we are always minimizing the weight of the paths from s to every other vertex, we will eventually "converge" to the (weight of) actual shortest paths.

We now formalize the idea above in the Bellman-Ford algorithm (in the following, we write the algorithm for directed graphs; if the original graph is undirected, simply turn it directed by adding both direction of each edge to the graph):

Bellman-Ford Algorithm: For finding single-source shortest paths from a given vertex s to all other vertices in weighted graph G:

1. Let $d[s] = 0$ and $d[v] = +\infty$ for $v \in V - \{s\}$.
2. For every edge $e = (u, v) \in E$:
   (a) If $d[v] > d[u] + w_e$, ***update*** $d[v] = d[u] + w_e$.
3. If no updates happen in the for-loop above, terminate and output $d[v]$ for every v as the shortest path distances; otherwise goto Line (2) and repeat.

Example: Let us consider the example above in the context of the Bellman-Ford algorithm.



- Before we run the for-loop in Line (2):

$$d[s] = 0, \qquad d[v_1] = +\infty, \qquad d[v_2] = +\infty, \qquad d[v_3] = +\infty.$$

- After the first time we run the for-loop in Line (2):

$$d[s] = 0, \qquad d[v_1] = 15 \text{ (updated via } (s, v_1)), \qquad d[v_2] = 3 \text{ (updated via } (s, v_2)), \qquad d[v_3] = +\infty.$$

- After the second time we run the for-loop in Line (2):

$$d[s] = 0, \qquad d[v_1] = 15, \qquad d[v_2] = 3, \qquad d[v_3] = 8 \text{ (updated via } (v_2, v_3)).$$

- After the third time we run the for-loop in Line (2):

$$d[s] = 0, \qquad d[v_1] = 12 \text{ (updated via } (v_3, v_1)), \qquad d[v_2] = 3, \qquad d[v_3] = 8.$$

- The fourth time we run the for-loop in Line (2) no update happens and hence we terminate with the above numbers which are the same as $dist(s, v)$ for all v in this example.

Proof of Correctness: The proof of consists of two parts: we prove (1) for all $v \in V$, $d[v] \geq dist(s, v)$ and (2) for all $v \in V$, $d[v] \leq dist(s, v)$ – this immediately implies $d[v] = dist(s, v)$ thus finalizing the proof.

The proof of first part is simple: whenever $d[v]$ is equal to some number $< +\infty$, we know that there is a sequence of edges $(u_k, v), (u_{k-1}, u_k), \cdots, (s, u_1)$ that were updated in the algorithm to eventually determine the value of $d[v] = \sum_{e:updated} w_e$. These edges thus form a path P_{sv} in the graph G and thus $d[v] = w(P_{sv})$, i.e., the weight of the path P_{sv}. Since $dist(s, v)$ is weight of the minimum weight path, $d[v] \geq dist(s, v)$.

We now prove the second (and the main) part. The proof is by induction. Let us define, for any integer $0 \leq i \leq n$, $dist_i(s, v)$ to be the minimum weight of any s-v path that uses *at most* i edges[1]. We prove by induction over i that for every $0 \leq i \leq n$, after the i-th time we run the for-loop in Line (2) of Bellman-Ford algorithm, $d[v] = dist_i(s, v)$ for all $v \in V$.

The base case for $i = 0$ holds since $dist_0(s, s) = 0$ and $dist_0(s, v) = +\infty$ for all $v \in V - \{s\}$ which is exactly the same as $d[s]$ and $d[v]$ before the first iteration (or "after" the 0-th iteration). Now suppose the induction hypothesis is true up to some integer i and we prove it for $i + 1$. Consider any vertex v and a s-v path P_{sv} with minimum weight from s to v that uses at most $i + 1$ edges so that $w(P_{sv}) = dist_{i+1}(s, v)$. Let u be the last vertex of this path immediately before v. We have that $w(P_{su}) = dist_i(s, u)$ (since otherwise we could switch the part from s to u with even a smaller weight path from s to u with at most i edges and find an even smaller weight path from s to v with at most $(i+1)$ edges, contradicting $w(P_{sv}) = dist_i(s, v)$). By induction hypothesis for i, this implies that $d[u] \leq dist_i(s, u)$ at the beginning of iteration $i + 1$ (or after iteration i). But by the update rule of the algorithm, after we visit the edge (u, v) in the for-loop, we definitely have

$$d[v] \leq d[u] + w_{uv} \leq dist_i(s, u) + w_{uv} = w(P_{su}) + w_{uv} = w(P_{sv}) = dist_{i+1}(s, v).$$

This proves the induction hypothesis for $i + 1$ as well (recall that $d[v]$ is non-increasing in the algorithm).

We are now done since $dist_{n-1}(s, v) = dist(s, v)$ for all $v \in V$ as any path can have at most $n - 1$ edges and thus after iteration $n - 1$ of the for-loop in the algorithm, by the induction hypothesis above, we have $d[v] \leq dist(s, v)$ for all $v \in V$. This completes part (2) of the proof.

Runtime Analysis: The proof of correctness above implies that the for-loop can be iterated at most $n-1$ times because after that $d[v] = dist(s, v)$ for all $v \in V$ and hence there can be no other update. Each iteration takes $O(m)$ time to go over all the edges so the total runtime is $O(nm)$.

We conclude this lecture by making some further remarks about the Bellman-Ford algorithm.

Remark. Bellman-Ford algorithm is actually a dynamic programming algorithm – in fact one of the very first dynamic programming algorithms ever (even though it may not look like it on the surface). You can read more about the dynamic programming view of the Bellman-Ford algorithm in Chapter 8 of Erickson's book (see Page 294 in particular).

Remark. Even though similar to before, we always assume that the weights on the edges are *positive* (or at least non-negative) in the shortest path problem, sometimes one may want to allow even negative weights

[1] For the example above, we have,

$$\begin{array}{llll} dist_0(s,s) = 0, & dist_0(s,v_1) = +\infty, & dist_0(s,v_2) = +\infty, & dist_0(s,v_3) = +\infty; \\ dist_1(s,s) = 0, & dist_1(s,v_1) = 15, & dist_1(s,v_2) = 3, & dist_1(s,v_3) = +\infty; \\ dist_2(s,s) = 0, & dist_2(s,v_1) = 15, & dist_2(s,v_2) = 3, & dist_2(s,v_3) = 8; \\ dist_3(s,s) = 0, & dist_3(s,v_1) = 12, & dist_3(s,v_2) = 3, & dist_3(s,v_3) = 8. \end{array}$$

Do you see the connection between these values and $d[v]$ computed after each for-loop of Bellman-Ford algorithm?

over the edges. For instance, consider a problem where going from a vertex to another vertex corresponds to the cost we need to pay for traversing that edge and hence shortest path between vertices gives us the minimum cost needed to go from the source vertex to any other vertex. However, one can imagine scenarios where traversing some edges can in fact brings us benefit (hence having a *negative* cost/weight).

Currently, there is *no efficient* algorithm known to solve the shortest path problem in graphs with negative edge weights in general (by efficient, think of a "fast" algorithm, a one that does not require exponential in m, n time which is too slow for any reasonable values of n and m – we will revisit this notion formally in a couple of lectures when we start studying P and NP problems). However, very interestingly, it turns out that the Bellman-Ford algorithm can be used even on graphs with negative edge weights as long as *we do not have a negative weight cycle*! This is because Bellman-Ford algorithm described above actually finds minimum weight *walks* and not paths in every graph even with negative edge weights (the proof above already implies this with minor modification). As long as we do not have a negative weight cycle, minimum paths and minimum walks coincide (there is no point in visiting a vertex multiple times and "paying" extra for it) and thus the algorithm also minimum weight paths. But when we have negative cycles, minimum walks simply find a negative cycle and iterate it infinitely so minimum weight walks have weight $-\infty$ and are different from minimum weight paths.

This concludes our study of the Bellman-Ford's algorithm.