

Shortest Path: Dijkstra's and Bellman-Ford

Lecturer: Debmalya Panigrahi

Scribe: Nat Kell, Tianqi Song, Tianyu Wang

1 Introduction

In this lecture, we will further examine shortest path algorithms. We will first revisit Dijkstra's algorithm and prove its correctness. Next, we will look at another shortest path algorithm known as the *Bellman-Ford* algorithm, that has a slower running time than Dijkstra's but allows us to compute shortest paths on graphs with negative edge weights.

2 Dijkstra's Correctness

In the previous lecture, we introduced Dijkstra's algorithm, which, given a positive-weighted graph $G = (V, E)$ and source vertex s , computes the shortest paths from s to all other vertices in the graph (you should look back at the previous lecture's notes if you do not remember the definition of the shortest path problem). As a refresher, here's the pseudo-code we saw for this algorithm:

Algorithm 1: DIJKSTRA'S ALGORITHM

Input: An undirected/directed graph $G = (V, E)$, a weight function $w : E \rightarrow \mathbb{R}^+$, and source vertex s .
Output: $d[v]$ stores the shortest path from s to v for all $v \in V$.

```

1  $d[s] \leftarrow 0$ 
2  $d[v] \leftarrow \infty$  for all  $v \neq s, v \in V$ 
3 Initialize a min-heap  $H \leftarrow V$  (where comparisons are done with  $d[v]$ ).
4 while  $H \neq \emptyset$  do
5    $u \leftarrow \text{DELETE-MIN}(H)$ 
6   for  $e = (u, v) \in E$  do
7      $\text{UPDATE}(u, v)$ 
8      $\text{DECREASE-KEY}(H, v, d[v])$ 
9 return  $d$ 
```

We define the subroutine UPDATE on line 7 to be:

$$\text{UPDATE}(u, v) : d[v] \leftarrow \min\{d[v], d[u] + w(u, v)\}. \quad (1)$$

(We will use this subroutine later on in the lecture for another algorithm, which is why we are defining it as a separate procedure). Informally, we think of $d[v]$ as our current estimate for the shortest path from s to v . The algorithm begins by initializing each $d[v] \leftarrow \infty$, except for our source vertex s , which we initialize so that $d[s] = 0$ (trivially, the shortest path from s to s is length 0). We then build a min-heap H on the vertex set that is organized based on these $d[\cdot]$ values. We proceed by repeatedly dequeuing vertex u with the minimum $d[u]$ value, and then we use this value to update the values of $d[v]$ for all vertices v that are adjacent to u .

To establish the correctness of Dijkstra's algorithm, we will argue that once we dequeue a vertex v from the heap, $d[v]$ stores the length of the shortest path from s to v .

Theorem 1. *When Dijkstra's algorithm terminates, $d[v]$ correctly stores the length of the shortest path from s to v .*

Proof. Denote $\text{SP}(s, v)$ to be the length of the shortest path from s to v in G . We will do a proof by contradiction: assume that there exists at least one vertex v such that $d[v] > \text{SP}(s, v)$ when v is removed from the heap. More specifically, let f be the first vertex to be removed from the heap that has this property. Let $P_f = \langle s = u_1, u_2, \dots, u_h = f \rangle$ be the sequence of vertices in the shortest path from s to f , where h is the number of vertices in this shortest path.

Consider the iteration of the while loop where we are about to dequeue f from the heap H (so we are about to execute line 5). We will deem a vertex v *shaded* if at this moment in the procedure v has already been removed from the heap. Denote u_k to be the first vertex along P_f that is unshaded, i.e., every vertex in subpath $\langle s, u_2, \dots, u_{k-1} \rangle$ is shaded and thus has been removed from the heap.

In the rest of the proof, we use the following two key observations:

Fact 1. *Since f is the first vertex that was removed from the heap such that at the time of its removal (and therefore at the end of the algorithm) we have $\text{SP}(s, f) < d[f]$, it follows that $\text{SP}(s, u_i) = d[u_i]$ for all $i < k$. In other words, since all the vertices in the subpath $\langle s, u_2, \dots, u_{k-1} \rangle$ were removed from the heap before f , each of these vertices must have correctly computed $d[\cdot]$ values.*

Fact 2. *For any $1 \leq i \leq j \leq h$, the subpath $\langle u_i, \dots, u_j \rangle$ of P_f must be a shortest path from u_i to u_j (otherwise, we could replace this subpath in P_f with a shorter path and as a result obtain a shorter path from s to f).*

Using these facts, we have two cases based on the value of k :

- Case 1: $k = h$, and so f is the first unshaded vertex in the path. Therefore, u_{h-1} is shaded and has been removed from the heap. By Fact 1, we know that $d[u_{h-1}] = \text{SP}(s, u_{h-1})$, i.e., the vertex immediately before f in P_f contains the correct $d[\cdot]$ value. Therefore, after we called $\text{UPDATE}(u_{h-1}, f)$ on the iteration where we removed u_{k-1} from the heap, $d[f]$ was set to be no larger than $\text{SP}(s, u_{h-1}) + w(u_{h-1}, f)$; however, since by definition P_f is the shortest path from s to f , it follows that $\text{SP}(s, u_{h-1}) + w(u_{h-1}, f) = \text{SP}(s, f)$. Thus, when we called $\text{UPDATE}(u_{k-1}, f)$, we did in fact store $\text{SP}(s, f)$ at $d[f]$. This is a contradiction since we originally assumed the algorithm would end with $d[f]$ storing a value larger than the length of the shortest path from s to f .
- Case 2: $k < h$, and so $u_k \neq f$ is an unshaded vertex along P_f that was not removed from the heap before this iteration; however, since u_k is the first unshaded vertex along the path, we know that u_{k-1} has been removed from the heap, and therefore by Fact 1, $d[u_{k-1}] = \text{SP}(s, u_{k-1})$.

Now, observe that on the iteration where we removed u_{k-1} from the heap, we called $\text{UPDATE}(u_{k-1}, u_k)$, implying that $d[u_k] \leq d[u_{k-1}] + w(u_{k-1}, u_k)$. We can then make the following argument:

$$\begin{aligned}
 d[u_k] &\leq d[u_{k-1}] + w(u_{k-1}, u_k) && \text{(since we called } \text{UPDATE}(u_{k-1}, u_k)) \\
 &= \text{SP}(s, u_{k-1}) + w(u_{k-1}, u_k) && \text{(using Fact 1: } u_{k-1} \text{ is shaded and } k-1 < k) \\
 &= \text{SP}(s, u_k) && \text{(by Fact 2: } \langle s, \dots, u_k \rangle \text{ is a subpath of } P_f) \\
 &< \text{SP}(s, f) && \text{(since edge weights are positive)} \\
 &< d[f] && \text{(by our original assumption).} \tag{2}
 \end{aligned}$$

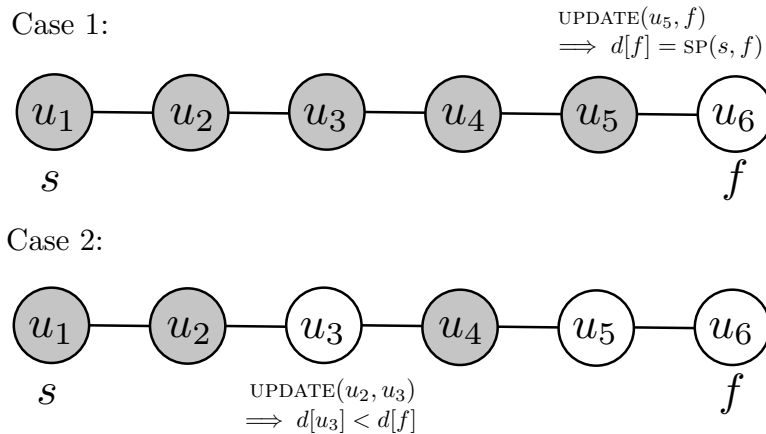


Figure 1: An illustration of the proof for Theorem 1 where $h = 6$. In Case 1, we know that every vertex except for f has been removed from the heap and therefore is shaded; because of this, we know that when $u_{h-1} = u_5$ was removed from the heap and called $\text{UPDATE}(u_5, f)$, the procedure correctly stored $\text{SP}(s, f)$ at $d[f]$ (which is a direct contradiction). In Case 2, there must be some first vertex along P_f (other than f) that has not been added to the heap yet; in the above example, this is vertex u_3 . This implies u_2 was removed from the heap and was removed before f , and therefore we know that the algorithm correctly stored $\text{SP}(s, u_3)$ at $d[u_3]$ when $\text{UPDATE}(u_2, u_3)$ was called. Since edge weights are positive, we can then make an argument that $d[u_3] < d[f]$, which contradicts the fact we are dequeuing f from the heap instead of u_3 .

Each of these steps should be (fairly) straightforward; we will point out that the second to last inequality is where our positive edge weight assumption comes in to play—because there must be at least one more edge in the path after u_k , the shortest path from s to u_k must be strictly smaller than the shortest path from s to f . Note that this is not true if we have negative edge weights; in fact, Dijkstra’s algorithm is incorrect when such edges exist in the graph (shortly, we will see algorithms capable of handling negative edge weights).

To complete the proof, note that the inequality implied by (2) gives us trouble: u_k is a vertex that is still in the heap, and yet we currently dequeuing vertex f even though inequality (2) implies that $d[u_k] < d[f]$. This provides us with our final desired contradiction.

□

We will end the section by making a couple remarks: Throughout the proof of Theorem 1, we are assuming that $d[v]$ can never contain a value that is strictly smaller than $\text{SP}(s, v)$ (otherwise some of the arguments above breakdown). However, we will argue in the next section that, regardless of the order we update $d[v]$ values and regardless of whether there are edges with negative weight, $d[v]$ can never contain a value that is shorter than the length of the shortest path to v (and since Dijkstra’s is just specifying a fancy order in which we do updates, we need not worry about this potential flaw in the above analysis).

Lastly, we will note that what we have presented here does not let us obtain the actual vertices in the shortest paths from s . However, by keeping track of the vertex that makes whose UPDATE call makes the final change to a vertex v ’s $d[v]$ value, we can figure out which vertex proceeds v in the shortest path from s to v ; therefore, we can follow these “predecessor” vertices back to s if we ever want to determine the actual paths. We will leave filling in these details as an exercise for you to complete on your own (and also note that such a modification can be made to the algorithms described in the next two sections, as well).

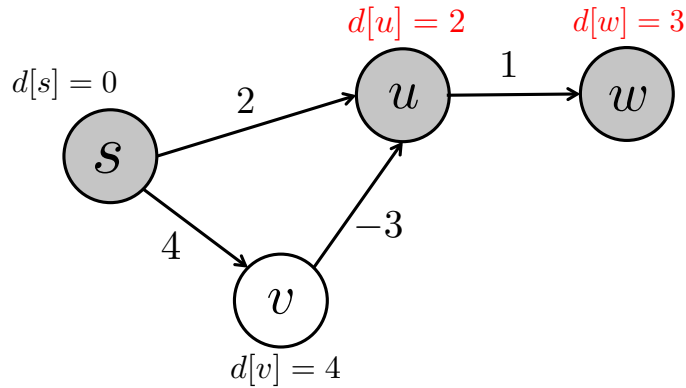


Figure 2: An directed graph showing why Dijkstra’s algorithm does not work when the graph contains negative edges. Note that shaded vertices again represent vertices that have been removed from the heap.

3 Dealing with Negative Edge Weights

3.1 The Trouble with Dijkstra’s

As mentioned above, we needed the assumption that all the edge weights are positive in order to prove Dijkstra’s correctness in Theorem 1. To make this more concrete, Figure 2 shows a small example where a negative edge causes Dijkstra’s Algorithm to fail. The example shows a snapshot in the algorithm’s execution where we have dequeued every vertex except for v (we are again shading vertices that have already been taken out of the heap).

In the argument we made in Theorem 1, the key property we showed is that when a vertex v is dequeued from the heap, its $d[v]$ value correctly stores the length of the shortest path from s to v . Observe that in this example, both u and w are violating this property—both of these vertices have been removed from the heap, but we can attain shorter paths for both vertices by first traveling to v and then taking the negative edge (v, u) . Even if we updated $d[u]$ to now be 1 after dequeuing v and calling UPDATE, $d[w]$ would still be incorrect. One can imagine that in larger graphs, there could be circumstances where we discover a shorter path that uses a negative edge, and this discovery affects the value of the of the shortest for many vertices that have already been removed from the heap; therefore, there is no small perturbation we can make to the definition of Dijkstra’s algorithm in order to fix this issue.

3.2 Doing the Updates Right

So how can we avoid this pitfall? One way to think of Dijkstra’s Algorithm is that we are just calling UPDATE(u, v) in some fancy order that is driven by the lengths of each shortest path. As a thought experiment, consider modifying Dijkstra’s to be this following (un-specifically) defined steps:

1. Set $d[s] = 0$ (where again s is our source).
2. Set $d[v] = \infty$ for all $v \neq s, v \in V$.
3. Run some sequence of UPDATE(u, v) procedures on the edges of E .

Again, note that Step 3 does not have to be the same order in which Dijkstra's algorithm would perform updates; there's also no requirement that we run an UPDATE on every edge. Although this process is useless as currently specified, we would like to show the following property about this procedure that holds true regardless of whether or not the the graph has negative edges.

Claim 1. *Again let $SP(s, v)$ denote the length of the shortest path from s to vertex v . If we execute steps 1, 2, and 3 as defined above, then $d[v] \geq SP(s, v)$ at the end of the process (no matter what sequence we specify in Step 3).*

Proof. To argue why this claim is true, we will use a very similar strategy we used in Theorem 1: Assume, for sake of contradiction, that f is the first vertex to violate this property, i.e., at some point during the procedure $d[f] < SP(s, f)$. Thus, there must have been some vertex u such that we called UPDATE(u, f), which resulted in $d[u] + w(u, f)$ being stored at $d[f]$ so that now $d[f] < SP(s, f)$. But f is the first violator of this property, and so it must be the case that $d[u] \geq SP(s, u)$ when this update is called. Combining these observations it follows that

$$\begin{aligned} d[f] &= d[u] + w(u, f) && \text{(since } u \text{ caused the violating update)} \\ &\geq SP(s, u) + w(u, f) && \text{(since } f \text{ is the first violator; } d[u] \geq SP(s, u)) \\ &\geq SP(s, f). \end{aligned} \tag{3}$$

To see why the last inequality is true, note that the shortest path from s to f has to travel through one of f 's neighboring vertices (i.e., any vertex v such that $(v, f) \in E$). u is a candidate for this vertex, and if it is the case that we travel through u on the shortest path from s to f , then the length of the this path is exactly $SP(s, u) + w(u, f)$ (can you see why? Look back at Fact 2 in Theorem 1). The only other case would be if there is some better path that uses a vertex other than u , but in this case we have that $SP(s, f) < SP(s, u) + w(u, f)$, which proves the inequality.

Of course, the inequalities from (3) show that $d[f] \geq SP(s, f)$, which contradicts the conditions we put on f and therefore establishes our claim. Observe that nowhere in this argument did we use any assumptions about the values of the edge weights, and so this claim is true even if the graph has negative edges. \square

Now, lets look a sequence for Step 3 that have a bit more structure (then with this structure, will use Claim 1 to argue that good things will happen). Specifically, consider a vertex $v \in V$ and let $P_u = \langle s = u_1, u_2, \dots, u_h = v \rangle$ denote the shortest path from s to v . Now suppose we guarantee that our update sequence looks like the following:

(some sequence of updates)
 UPDATE(s, u_2)
 (another sequence of updates)
 UPDATE(u_2, u_3)
 (another sequence of updates)
 \vdots
 UPDATE(u_{h-2}, u_{h-1})
 (another sequence of updates)
 UPDATE(u_{h-1}, v)

So informally, the structure we are looking at ensures that there is a subsequence of the updates such that they happen along the edges of P_v . Note that any sequence of updates can occur in the places labeled “(another sequence of updates)”.

Therefore, using an argument that is much akin to the ones we made in Theorem 1 and Claim 1, we can argue that when UPDATE(u_{i-1}, u_i) is called in the above sequence for some $1 \leq i < h$, $d[u_i]$ now stores the length shortest path from s to u_i . Specifically, we inductively assume $d[u_{i-1}]$ stores SP(s, u_{i-1}) after UPDATE(u_{i-2}, u_{i-1}) has been called. Claim 1 guarantees that during the “(another sequence of updates)” before the call to UPDATE(u_{i-1}, u_i) we will not somehow ruin the value stored at $d[u_{i-1}]$ by storing an invalid path length there that is lower than SP(s, u_{i-1}). Therefore, we will correctly transfer the length of the shortest path from s to u_i into $d[u_i]$ when we call UPDATE(u_{i-1}, u_i).

3.3 Putting it Together: The Bellman-Ford Algorithm

By the above arguments, if we can define a sequence for Step 3 such that we guarantee such a subsequence exists for every vertex v in the graph, we are done! The above argument ensures that the length of the shortest path to v will propagate to $d[v]$. Defining such a sequence might seem difficult, but the algorithm we’ll define will just used brute forced: *we will do an update for every edge in the graph, and then repeat this $n - 1$ times*. The reason we this works is almost immediate; this sequence contains every possible sub-sequence of updates that could potentially define a path in the graph (which is why we need $n - 1$ repetitions). Thus, we are guaranteed not to miss any subsequences that define the edges of the shortest path from s to some vertex v .

This approach finally gives an algorithm, which is formally outlined in Algorithm 2 and is known as the *Bellman-Ford* algorithm.

The running time of Bellman-Ford is in $O(nm)$ since we are updating every edge in the graph $n - 1$ times. This is slower than Dijkstra’s Algorithm, which runs in $O((n + m) \log n)$ time, but now we can deal with negative edges. There are also many other interesting tradeoffs between Bellman-Ford and Dijkstra’s that manifest when these algorithms are examined in certain settings (e.g. computer networks); however, such tradeoffs are outside the scope of what we will discuss in this class.

Finally note that, even though Bellman-Ford can handle negative edge weights, we still must assume that the given graph contains no *negative cycles*. If such cycles exist, there are no shortest paths in the graph since from s we can travel to this cycle and loop around forever to continually obtain shorter and shorter

Algorithm 2: THE BELLMAN-FORD ALGORITHM

Input: An undirected/directed graph $G = (V, E)$, a weight function $w : E \rightarrow \mathbb{R}$, and a source vertex s .
Output: $d[v]$ stores the shortest path from s to v for all $v \in V$.

- 1 $d[s] \leftarrow 0$
- 2 $d[v] \leftarrow \infty$ for all $v \neq s, v \in V$
- 3 **for** $i = 1$ to $n - 1$ **do**
- 4 **for each** $(u, v) \in E$ **do**
- 5 UPDATE(u, v)
- 6 **return** d

paths.

Thus, one might ask the question: how can we detect if there is a negative cycle the given graph? In fact, we can answer this question by running Bellman-Ford for one extra iteration.

3.3.1 Application: Negative Cycle Detection

The idea for negative cycle detection is straightforward. We run Bellman-Ford for one more iteration and see whether any shortest path is updated.

Algorithm 3: NEGATIVE CYCLE DETECTION

Input: An undirected/directed graph $G = (V, E)$, a weight function $w : E \rightarrow \mathbb{R}$.
Output: *True* if there is a negative cycle; *False* otherwise.

- 1 Run Bellman-Ford on G w.r.t. any source vertex s to get $d[v]$ for all $v \in V$
- 2 **for each** $(u, v) \in E$ **do**
- 3 **if** $d[u] + w(u, v) < d[v]$ **then**
- 4 **return** *True*
- 5 **return** *False*
