COMPSCI 330: Design and Analysis of Algorithms 2/16/2016

Shortest Path

Lecturer: Debmalya Panigrahi *Scribe: Nat Kell, Tianqi Song*

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. Lastly, we look at an $O(n^3)$-time algorithm for computing the shortest paths between all pairs of vertices in a graph (recall that for the former two algorithms, we only compute the shortest paths from a particular source vertex).[1]

2 Dijkstra's Correctness

In the last 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 $w : E \to \mathbb{R}^+$, 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     $u \leftarrow$ DELETE-MIN$(H)$
6     for $e = (u,v) \in E$ do
7         UPDATE$(u,v)$
8         DECREASE-KEY$(H, v, d[v])$
9 return $d$
```

We define the subroutine UPDATE on line 7 to be:

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

(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

[1]This note is originally written by Nat Kell for this class in Fall 2014. Tianqi Song edited this note.

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 the 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 $\mathrm{SP}(s,v)$ 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 $d[v] > \mathrm{SP}(s,v)$ 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 $P_f = \langle s = u_1, u_2, \ldots, u_h = f\rangle$ 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 $\langle s, u_2, \ldots, u_{k-1}\rangle$ 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 $\mathrm{SP}(s,f) < d[f]$, it follows that $\mathrm{SP}(s,u_i) = d[u_i]$ for all $i < k$. In other words, since all the vertices in the subpath $\langle s, u_2, \ldots, u_{k-1}\rangle$ were removed from the heap before f, each of these vertices must have correctly computed $d[\cdot]$ values.*

Fact 2. *For any $1 \le i \le j \le h$, the subpath $\langle u_i, \ldots, u_j\rangle$ 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 $d[u_{h-1}] = \mathrm{SP}(s,u_{h-1})$, i.e., the vertex immediately before f in P_f contains the correct $d[\cdot]$ value. Therefore, after we called $\textsc{update}(u_{h-1}, f)$ on the iteration where we removed u_{k-1} from the heap, $d[f]$ was set to be no larger than $\mathrm{SP}(s,u_{h-1}) + w(u_{h-1}, f)$; however, since by definition P_f is the shortest path from s to f, it follows that $\mathrm{SP}(s,u_{h-1}) + w(u_{h-1}, f) = \mathrm{SP}(s,f)$. Thus, when we called $\textsc{update}(u_{k-1}, f)$, we did in fact store $\mathrm{SP}(s,f)$ 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 , $d[u_{k-1}] = \mathrm{SP}(s,u_{k-1})$.

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

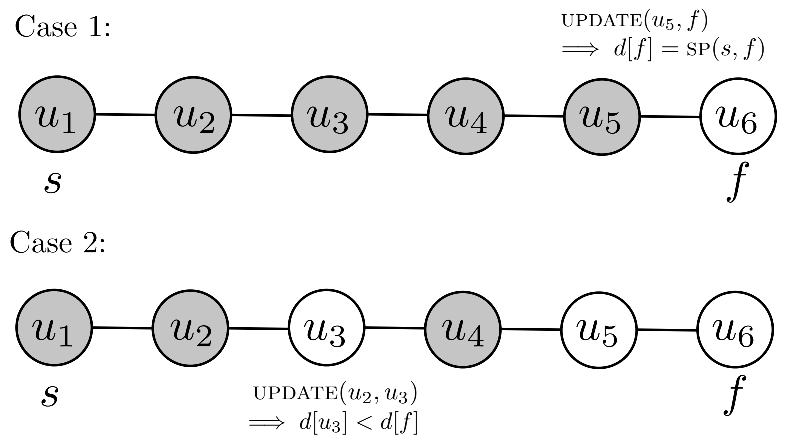

Figure 1: An illustration of the proof for Theorem 1 where $h = 6$. In Case 1, we know that every vertex except for f has been removed from the heap and therefore is shaded; because of this, we know that when $u_{h-1} = u_5$ was removed from the heap and called UPDATE(u_5, f), the procedure correctly 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 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 dequeueing f from the heap instead of u_3.

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

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

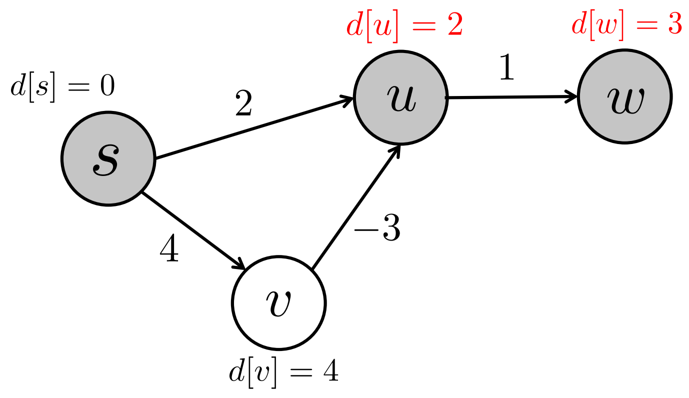

Figure 2: An directed graph showing why Dijkstra's algorithm does not work when the graph contains negative edges. Note that shaded vertices again represent vertices that have been removed from the heap.

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).

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 $\text{UPDATE}(u,v)$ 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 $\text{UPDATE}(u,v)$ 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* $\text{SP}(s,v)$ *denote the length of the shortest path from s to vertex v. If we execute steps 1, 2, and 3 as defined above, then* $d[v] \geq \text{SP}(s,v)$ *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 $d[f] < \text{SP}(s,f)$. Thus, there must have been some vertex u such that we called $\text{UPDATE}(u,f)$, which resulted in $d[u] + w(u,f)$ being stored at $d[f]$ so that now $d[f] < \text{SP}(s,f)$. But f is the first violator of this property, and so it must be the case that $d[u] \geq \text{SP}(s,u)$ when this update is called. Combining these observations it follows that

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

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 $\text{SP}(s,u) + w(u,f)$ (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 $\text{SP}(s,f) < \text{SP}(s,u) + w(u,f)$, which proves the inequality.

Of course, the inequalities from (3) show that $d[f] \geq \text{SP}(s,f)$, 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 $P_u = \langle s = u_1, u_2, \ldots, u_h = v \rangle$ denote the shortest path from s to v. Now suppose we guarantee that our update sequence looks like the following:

$$\begin{gathered}\text{(some sequence of updates)}\\ \textsc{Update}(s,u_2)\\ \text{(another sequence of updates)}\\ \textsc{Update}(u_2,u_3)\\ \text{(another sequence of updates)}\\ \vdots\\ \textsc{Update}(u_{h-2},u_{h-1})\\ \text{(another sequence of updates)}\\ \textsc{Update}(u_{h-1},v)\end{gathered}$$

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 $\textsc{Update}(u_{i-1},u_i)$ 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 $\text{SP}(s,u_{i-1})$ after $\textsc{Update}(u_{i-2},u_{i-1})$ has been called. Claim 1 guarantees that during the "(another sequence of updates)" before the call to $\textsc{Update}(u_{i-1},u_i)$ we will not somehow ruin the value stored at $d[u_{i-1}]$ by storing an invalid path length there that is lower than $\text{SP}(s,u_{i-1})$. Therefore, we will correctly transfer the length of the shortest path from s to u_i into $d[u_i]$ when we call $\textsc{Update}(u_{i-1},u_i)$.

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

```
Algorithm 2: THE BELLMAN-FORD ALGORITHM
  Input: An undirected/directed graph $G = (V, E)$, a weight function $w : E \to \mathbb{R}$, 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 $d$
```

since from s we can travel to this cycle and loop around forever to continually obtain shorter and shorter paths.

Thus, one might ask the question: how can we detect if there is a negative cycle the given graph? We claim that that it is possible to answer this question by running Bellman-Ford for one extra iteration—we will leave the details of this modification as an exercise.

4 Computing Shortest Paths for All Pairs

In the previous two sections, we were just interested in fixing a single source vertex s and then computing all the shortest paths from s to all other vertices in the graph; however, one can imagine circumstance where we want to readily know the shortest paths between any pair of vertices and not just a from a particular source. Thus in this section, we will look at an algorithm that returns a n^2 size array d, where $d[u, v]$ stores the length of the shortest path from vertex u to vertex v in G.

In a graph with negative edges, a naive way of solving this problem is to run Bellman-Ford n separate times where each time we use a different vertex as the source. Since Bellman-Ford's running time is in $O(nm)$, this yields a $O(n^2 m)$-time algorithm. Note that it is possible for $m = \Omega(n^2)$, so in the worst case this is an $O(n^4)$-time algorithm. In this section, we will give an algorithm know as the *Flyod-Warshall* algorithm, which improves on this bound with a running time of $O(n^3)$.

To define this algorithm, we first fix an ordering on the vertices and then label them $1, 2, \ldots, n$ based on this ordering. Next we define the function PATH(i, j, k) as follows:

$$\textsc{Path}(i,j,k) = \begin{array}{l}\text{The length of the shortest path from } i \text{ to } j \text{ that only}\\ \text{uses vertices } \{1,\ldots,k\} \text{ as intermediate vertices.}\end{array}$$

Note that by "intermediate" vertices, we mean any vertex other than i and j that we use along the path. For our base cases, we will define PATH$(i, j, 0)$ for all pairs of vertices i and j to be:

$$\textsc{Path}(i,j,0) = \begin{cases} w(i,j) & \text{if } (i,j) \in E \\ \infty & \text{otherwise.} \end{cases}$$

Intuitively this makes sense. Since $k = 0$, we are defining the length of the shortest path between i and j that uses no intermediate vertices. Thus, if i and j are adjacent in G, then the length of the this path is the weight of edge (i,j); otherwise, there is no way to reach j from i without using other vertices, and so we set this length to ∞.

Now we will inductively define $\textsc{Path}(i,j,k)$. Let P be the shortest path between i and j that only uses vertices $\{1,\dots,k\}$ as intermediate vertices. There are two cases: either vertex k belongs to P or it does not. If P does not include vertex k, then we just have $\textsc{Path}(i,j,k) = \textsc{Path}(i,j,k-1)$; in this case, P is also a shortest path from i to j that only uses vertices 1 through $k-1$, and so these two path lengths must be identical.

Otherwise, we know that we must arrive at vertex k somewhere along P. Thus, we can break P up into two parts: we first must take some subpath from i to k; call this subpath $P_1 = \langle i = u_1, u_2, \dots, u_h = k\rangle$. After we arrive at k on P, we then take another subpath from k to j; call this path $P_2 = \langle k = u_h, u_{h+1}, \dots, u_{r-1}, u_r = j\rangle$.

We can then make the following observation: the lengths of the P_1 and P_2 are $\textsc{Path}(i,k,k-1)$ and $\textsc{Path}(k,j,k-1)$, respectively. It should be clear that $\{u_2,\dots,u_{h-1},u_{h+1},\dots,u_{r-1}\}$ is a subset of $\{1,\dots,k-1\}$. P was originally restricted to the set of vertices up to k, and since no shortest path can repeat a vertex (this would create a cycle that could then be removed to shorten the path), it follows that the intermediate vertices along P other than k must be limited to the set $\{1,\dots,k-1\}$. We also know that P_1 and P_2 must be the *shortest* paths that use this limited set of vertices; otherwise, we could replace either P_1 or P_2 with better subpaths and shorten the length of P (again, this is the same argument made by Fact 2 in Theorem 1).

By the above analysis, we can define $\textsc{Path}(i,j,k)$ for $k \geq 1$ as follows.

$$\textsc{Path}(i,j,k) = \min\{\textsc{Path}(i,j,k-1), \textsc{Path}(i,k,k-1) + \textsc{Path}(k,j,k-1)\}. \tag{4}$$

Here, we are just taking the minimum between the two possible cases described above (whichever value in the min is smaller tells us what case we are in). This in turn gives us an algorithm: for each $k = 1$ up to n, we compute $\textsc{Path}(i,j,k)$ and store this value in a table. After this procedure, we can return $\textsc{Path}(i,j,n)$ for our $d[i,j]$ entry, since this entry will store the shortest path from i to j that is allowed to use any vertex as an intermediate.

Since $\textsc{Path}(i,j,k)$ is only defined in terms of values of $\textsc{path}$ that have smaller k values (namely $k-1$), we are guaranteed to have already stored each of the necessary entries in our $\textsc{path}$ table whenever computing $\textsc{Path}(i,j,k)$; therefore, it will take constant time to compute the entry for a given (i,j,k) triple. Since the domain for each parameter is from 1 to n, there are $O(n^3)$ $\textsc{Path}(i,j,k)$ entries we have to compute in total; thus, the algorithm runs in $O(n^3)$ time.