CS31 (Algorithms), Spring 2020 : Lecture 12

Date:

Topic: Graph Algorithms 3: BFS, Dijkstra

Disclaimer: These notes have not gone through scrutiny and in all probability contain errors. Please discuss in Piazza/email errors to deeparnab@dartmouth.edu

---

In this lecture and next, we look at shortest paths in graphs. Graphs will now have *costs* on edges and shortest paths will be with respect to these costs. We begin with the simplest case: when all the costs are 1.

1 Breadth First Search

In DFS from a vertex, that is the algorithm DFS(G, v), we explored unvisited neighbors but as soon as we discovered one such vertex we recursively started our search from that vertex again. In some sense, the algorithm tries to go as "deep" as possible and then backtracks if it can't dig deeper.

Breadth First Search, or BFS, takes the broader approach. Starting from a vertex, it first traverses all its unvisited neighbors putting them in a queue. Subsequently, we visit all neighbors of these neighbors, and so on and so forth. The plus side of BFS is that it seems to reach a vertex "quickly" (and we make this precise in a bit) while DFS could meander and perambulate before reaching any particular vertex. The description of BFS I give below is a bit more "complicated" than usual definitions; but it will be readily generalize to the setting with costs on edges.

The algorithm takes input a graph (directed or undirected) G and a "source" vertex s. At the end of it, it assigns a *distance* label $\mathsf{dist}[v]$ to every vertex $v \in V$. $\mathsf{dist}[v]$ indicates the length of the shortest path obtained from s to v so far; if no such path found, this is set to ∞. Thus, initially $\mathsf{dist}[s] = 0$ and all other dists are ∞. Subsequently, everytime a vertex v is being processed, it ***updates*** the distance label of every out-neighbor u whose $\mathsf{dist}[u] > \mathsf{dist}[v] + 1$ to just $\mathsf{dist}[v] + 1$. If u's distance label is changed, it is put in the queue. This update step is crucial to *all* known shortest path algorithms. So much so, that it is worth thinking of it is a subroutine.

```
1:  procedure BFS($G, s$):
2:      ▷ Returns a distance label to every vertex.
3:      ▷ Every vertex not s also has a pointer parent to another vertex.
4:      $\mathsf{dist}[s] \leftarrow 0$; $\mathsf{dist}[v] \leftarrow \infty$ otherwise.
5:      $\mathsf{parent}[v] \leftarrow \perp$ for all $v$.
6:      Assign queue $Q$ initialized to $s$.  ▷ FIFO Queue
7:      while $Q$ is not empty do:
8:          $v = Q.\mathsf{remove}()$.  ▷ The first entry in $Q$
9:          for all neighbors $u$ of $v$ do:  ▷ Update $(v, u)$.
10:             if ($\mathsf{dist}[u] > \mathsf{dist}[v] + 1$) then:  ▷ $\mathsf{dist}[u]$ too large.
11:                 ▷ For BFS, in fact, $\mathsf{dist}[u] = \infty$.
12:                 Set $\mathsf{dist}[u] = \mathsf{dist}[v] + 1$.  ▷ Update $\mathsf{dist}[u]$
13:                 $Q.\mathsf{add}(u)$.  ▷ Since $u$'s distance label was modified, put it in the $Q$
14:                 Set $\mathsf{parent}[u] = v$.
```

We start by making trivial but important observations.

Observation 1. $\mathsf{dist}[v]$ of any vertex can never increase over time. Also, $\mathsf{dist}[v]$ is a non-negative integer (unless it is infinity). Finally, $\mathsf{dist}[v] = \mathsf{dist}[\mathsf{parent}[v]] + 1$ or ∞.

Proof. (Sketch) We modify $\mathsf{dist}[u]$ to $\mathsf{dist}[v] + 1$ only if it was bigger than $\mathsf{dist}[v] + 1$. Furthermore, it is set by adding 1 to $\mathsf{dist}[v]$. If $\mathsf{dist}[v]$ was an integer, then so is $\mathsf{dist}[u]$. The last line follows from Line 12 and Line 14 □

Remark: *The above observation gives the following intuition about the distance labels* $\mathsf{dist}[v]$ *which is important to have. Note that by repeatedly taking parents, we get that the reverse of the path* $(v, \mathsf{parent}[v], \mathsf{parent}[\mathsf{parent}[v]], \ldots, s)$ *is a path from s to v of length precisely* $\mathsf{dist}[v]$. *Therefore, at any point of time,* $\mathsf{dist}[v]$ *is an* ***upper bound*** *on the shortest length path from s to v. And every update step on an edge (v, u) possibly updates the distance label on the vertex u.*

Lemma 1. (Monotonicity Lemma.) Fix a particular while loop, and let $Q = [u_1, \ldots, u_k]$ be the queue content just before the beginning of the loop. Then $\mathsf{dist}[u_1] \le \mathsf{dist}[u_2] \le \cdots \le \mathsf{dist}[u_k] \le \mathsf{dist}[u_1] + 1$.

Proof. The proof is by induction over the while loops. At the beginning of the first while loop, $Q = [s]$ and the lemma is vacuously true. Otherwise, fix a while loop, and let $Q = [u_1, \ldots, u_k]$, and let the lemma be true right now. We now show it remains true after this loop.

Let us see what the while loop does. First, it removes the vertex u_1 from Q. It will then add *every* neighbor x with current $\mathsf{dist}[x] > \mathsf{dist}[u_1]+1$, and upon adding, the distances become $\mathsf{dist}[x] = \mathsf{dist}[u_1]+1$. Let these neighbors be $x_1, \ldots, x_r$ (note r could be 0 and there could be no such neighbors). After this while loop is run, the contents of the Q is precisely $[u_2, u_3, \ldots, u_k, x_1, \ldots, x_r]$. By induction, we know $\mathsf{dist}[u_2] \le \ldots \le \mathsf{dist}[u_k]$. Furthermore, since $\mathsf{dist}[u_k] \le \mathsf{dist}[u_1] + 1$ (by induction), we see $\mathsf{dist}[u_k] \le \mathsf{dist}[x_i]$ for all $1 \le i \le r$. Finally, $\mathsf{dist}[x_i] = \mathsf{dist}[u_1] + 1 \le \mathsf{dist}[u_2] + 1$. And thus, we get $\mathsf{dist}[u_2] \le \ldots \le \mathsf{dist}[u_k] \le \mathsf{dist}[x_1] \le \ldots \le \mathsf{dist}[x_r] \le \mathsf{dist}[u_2] + 1$. □

The above lemma has a bunch of very useful corollaries.

Corollary 1. In the BFS algorithm, a vertex v never enters the queue more than once.

Proof. Suppose not, and suppose v is the first vertex which enters Q twice. Let u be the first vertex which added v to Q the first time, and let x be the second vertex which added v the second time into Q. When u added v to Q, the algorithm set $\mathsf{dist}[v] = \mathsf{dist}[u] + 1$. Now consider the time x is adding v again to Q. At this point, we must have $\mathsf{dist}[v] > \mathsf{dist}[x] + 1$. That is, $\mathsf{dist}[u] > \mathsf{dist}[x]$. But since u was popped out of the queue before x, the previous monotonicity lemma tells us $\mathsf{dist}[x] \ge \mathsf{dist}[u]$. Contradiction. Therefore, no vertex enters the Q twice. □

Corollary 2 (BFS runtime)**.** BFS(G, s) runs in $O(n + m)$ time.

Proof. Every while loop pops a vertex v out and then scans all its out-neighbors. That is, this while loop takes $O(\deg^+(v))$ time. By Corollary 1, the vertex v is processed at most once. Thus, the time taken is $\sum_{v \in V} O(\deg^+(v)) = O(n + m)$ (as in DFS). □

Now comes the main lemma which connects shortest paths and BFS. I would like to *stress* that the next lemma doesn't need any of the previous lemmas, but *holds* as long as the BFS terminates to give distance labels.

Lemma 2 (Shortest Path Lemma)**.** Let $p = (s = x_0, x_1, \ldots, x_k)$ be any path in the graph. Then $\mathsf{dist}[x_i] \le i$ for all $0 \le i \le k$.

Proof. Suppose not, and let i be the smallest index for which this is false (note that the statement is true for 0 and therefore $i \ge 1$). Thus, $\mathsf{dist}[x_{i-1}] \le (i-1)$ but $\mathsf{dist}[x_i] > i$. Now consider the time x_{i-1} is removed from the Q (since its dist is not infinity, it must have entered Q and thus must have left). In this while loop, the for-loop will check if $\mathsf{dist}[x_i] > \mathsf{dist}[x_{i-1}] + 1 \ge i$. Now, $\mathsf{dist}[x_i] > i$ at the end of the algorithm, and so by Observation 1, $\mathsf{dist}[x_i] > i$ in this while loop as well. Therefore, the "if" statement will succeed, and then it would set $\mathsf{dist}[x_i] = \mathsf{dist}[x_{i-1}] + 1 \le i$. But this contradicts that $\mathsf{dist}[x_i] > i$ in the end. □

Theorem 1. Using BFS once can find the shortest path length from a vertex s to every other reachable vertex v in $O(n+m)$ time.

Proof. We claim $\mathsf{dist}[v]$ is indeed the shortest path length from s to v. Lemma 2 shows that $\mathsf{dist}[v]$ is at most the shortest path length. All that remains to show is that there is a path from s to v of exactly this length. Indeed, it is the *reverse* of $(v, \mathsf{parent}[v], \mathsf{parent}[\mathsf{parent}[v]], \ldots, s)$. □

The Shortest Path Tree. Note that $\mathrm{BFS}(G, s)$ returns a tree rooted at s defined by the parents. More precisely, consider the graph on all vertices v with $\mathsf{dist}[v] < \infty$ where we add the edges $(\mathsf{parent}(v), v)$. This tree is called the *shortest path tree*. For every vertex v which is reachable from s, as Lemma 2 shows, the unique path from s to v in the tree is a shortest path.

"Weighted" BFS. We now want to generalize to the case when edges of the graphs have weights or costs. That is, every edge (u, v) has a cost $c(u, v)$. Imagine for now these costs are non-negative integers. How would we *generalize* the above algorithm? Well, one "obvious" generalization seems the following. The changes are marked in red.

```
1:  procedure wBFS($G, s$):
2:      $\mathsf{dist}[s] = 0$; $\mathsf{dist}[v] = \infty$ otherwise.
3:      $\mathsf{parent}[v] = \perp$ for all $v \neq s$.
4:      Assign queue $Q$ initialized to $s$.
5:      while $Q$ is not empty do:
6:          $v = Q.\mathsf{remove}()$.
7:          for all neighbors $u$ of $v$ do:
8:              if ($\mathsf{dist}[u] > \mathsf{dist}[v] + c(v, u)$) then:
9:                  Set $\mathsf{dist}[u] = \mathsf{dist}[v] + c(v, u)$.
10:                 $Q.\mathsf{add}(u)$.
11:                 Set $\mathsf{parent}[u] = v$.
```

What's "wrong" with the above algorithm? Well, first let us show what is "right" about the algorithm. The next lemma shows that ***if*** the above algorithm terminates, then it gives the correct answer in that the dist labels are lower bounds on the shortest path. First we begin by noting that Observation 1 still holds (check this).

Lemma 3 (Shortest Path Lemma in Weighted Graphs)**.** ***Suppose*** the algorithm wBFS terminates with distance labels $\mathsf{dist}[v]$ on every vertex. Let $p = (s = x_0, x_1, \ldots, x_k)$ be any path in the graph. Then $\mathsf{dist}[x_i] \leq \sum_{j=0}^{i-1} c(x_j, x_{j+1})$ for all i, that is, the cost of the path from s to i. This is true ***even*** if the costs are negative.

Proof. The proof is almost a rewriting of the proof of Lemma 2.

For brevity, let us use C_i to denote the cost $\sum_{j=0}^{i-1} c(x_j, x_{j+1})$. We need to show $\mathsf{dist}[x_i] \leq C_i$ for all i. Suppose not, and let i be the smallest index for which this is false (note it is true for $i = 0$.) Thus, $\mathsf{dist}[x_{i-1}] \leq C_{i-1}$ but $\mathsf{dist}[x_i] > C_i$. Since $\mathsf{dist}[x_{i-1}]$ is finite, its distance label was set, and thus x_{i-1} was added to the Q at least once. Now consider the time x_{i-1} is removed from the Q *for the last time* (we are not claiming any more that every vertex enters Q at most once). In this while loop, the for-loop will check if $\mathsf{dist}[x_i] > \mathsf{dist}[x_{i-1}] + c(x_{i-1}, x_i) \geq C_{i-1} + c(x_{i-1}, x_i) = C_i$. Now, we have supposed that $\mathsf{dist}[x_i] > C_i$ at the end of the algorithm, and since distances don't increase, $\mathsf{dist}[x_i] > C_i$ at this while loop as well. Therefore, the "if" statement will succeed, and then it would set $\mathsf{dist}[x_i] = \mathsf{dist}[x_{i-1}] + c(x_{i-1}, x_i) \leq C_{i-1} + c(x_{i-1}, x_i) = C_i$. But this contradicts that $\mathsf{dist}[x_i] > C_i$ in the end. □

Ok, but that ***if*** is a big if. Look at the algorithm wBFS again: it has a *while* loop. Whenever you see a while loop, one needs to be concerned about the running time. For BFS, we took care of this first (Corollary 2). What about the wBFS problem? Before reading the next paragraph, take some time to think about what can you say about the running time of wBFS. Can it do an infinite loop? Why can't we say that wBFS runs in $O(n + m)$ time as in Corollary 2? Well, that proof needed Corollary 1. Can you see why Corollary 1 ***fails*** for wBFS? Implement this algorithm up and see if it terminates on graphs of your choice.

Lemma 4. If all the costs are non-negative, then the algorithm wBFS terminates.

Proof. To show an algorithm with a while loop terminates, one usual way is to find a "measure" which keeps decreasing as the while loop proceeds. What is one possibility here? Well, we already know that the distance labels cannot increase. So, let us use D to denote $\sum_v \mathsf{dist}[v]$ for all vertices v. Now if some vertex has $\mathsf{dist}[v] = \infty$, we have $D = \infty$. To take care of this, let's just use $C = \max_{u,v} c(u, v)$ and initially set $\mathsf{dist}[v] = (n-1)C + 1$ since the total cost of any path is $\leq (n-1)C$. Therefore, in the beginning, $D \leq (n-1)^2 C$. Now we see that in every while loop either the size of the Q decreases, or D decreases by at least 1. This follows from Observation 1; if we change the distance, then we drop it by at least 1. Therefore, in $O(n^2 C)$ while loops the algorithm does terminate. □

Remark: *Can you come up with an example of a graph with non-negative costs where the above algorithm* does *take longer than $O(n + m)$ time? This is not trivial. One way to find would be to figure out why Corollary 1 was false, etc. That is, the same vertex can enter the Q multiple times. After seeing the example, does it suggest a way to improve the above algo?*

Remark: *If the costs are allowed to be negative integers, then can you come up with an example where the algorithm* ***does not*** *terminate?*

2 Dijkstra's Algorithm

SHORTEST PATH
Input: Graph $G = (V, E)$. Costs $c(e) > 0$ on every edge $e \in E$. Source vertex s.
Output: Shortest Paths to every vertex $v \in V$.

In this section, we show that one can make the algorithm described above "efficient" by not putting *every* eligible neighbor of the vertex in consideration into the queue. This is the famous DIJKSTRA's algorithm named after its inventor, Edsger Dijkstra. There are two differences to WBFS. One, by *design* it sets the distance label at most once. This is done by updating the distance label only over neighbors which have not their distance labels set. Two, and this is key, it only adds *one* vertex to the queue, but which one is crucial: it picks the one with the *smallest* dist$[v]$.

```
1:  procedure DIJKSTRA($G, s$):
2:      ▷ Returns a distance label to every vertex. Every vertex not s also has a pointer parent to another vertex.
3:      dist$[s] = 0$; dist$[v] = \infty$ otherwise.
4:      parent$[v] = \perp$ for all $v \neq s$.
5:      Initialize $Q = s$.
6:      Initialize $R = \emptyset$.  ▷ $R$ will be the "reached" vertices whose dist$[v]$'s have been set.
7:      while $Q$ is not empty do:
8:          $v \leftarrow Q$.remove()
9:          $R \leftarrow R + v$
10:         for all neighbors $u$ of $v$ not in $R$ do:  ▷ The distance labels set for only vertices outside $R$
11:             if (dist$[u]$ > dist$[v] + c(v, u)$) then:
12:                 Set dist$[u] \leftarrow$ dist$[v] + c(v, u)$.
13:                 Set parent$[u] \leftarrow v$.
14:         Find $u$ with minimum dist$[u]$ among vertices not in $R$.
15:         ▷ This is the key time improvement step.
16:         $Q$.add($u$).
```

Remark: *As one can see, the Q is not even needed above. Indeed, the typical exposition of* DIJKSTRA *(which you may have seen in CS10) contains only the R.*

Lemma 5 (Termination)**.** The While loop in DIJKSTRA runs for at most $n + 1$ iterations.

Proof. At each while-loop iteration, a vertex from Q is deleted, added into R, and a vertex not in R, if any, is added. The size of Q remains 1 till R becomes the whole vertex set, in which case the Q becomes $\emptyset$ and the while loop terminates. Since the size of R precisely increases by one, and the while loop terminates after $\leq n + 1$ iterations. □

Lemma 6. Once a vertex x enters R its distance label dist and parent is never changed again. Furthermore, dist$[x]$ = dist$[$parent$[x]] + c($parent$[x], x)$.

Proof. This is made sure by Line 10 since it updates distances, if at all, only for vertices not in R. □

Lemma 7 (Monotonicity Lemma)**.** Let the order in which vertices are added to R be $(v_0, v_1, \ldots, v_n)$. Note that $v_0 = s$ itself. Then,

$$\mathsf{dist}[v_0] \le \mathsf{dist}[v_1] \le \cdots \le \mathsf{dist}[v_n]$$

where $\mathsf{dist}[]$ are the distance labels at the end of the algorithm.

Proof. Suppose not, and let v_{i+1} be the first vertex for which this violated. That is, $\mathsf{dist}[v_{i+1}] < \mathsf{dist}[v_i]$. Since v_i was added to R before v_{i+1}, Line 14 implies the time when v_i is added to R, we have $\mathsf{dist}[v_i] \le \mathsf{dist}[v_{i+1}]$. Also note that by definition, v_{i+1} is the vertex added in the next iteration.

Now, if (v_i, v_{i+1}) is *not* an edge then after v_i enters Q, $\mathsf{dist}[v_{i+1}]$ is not modified in that for-loop. Thus when v_{i+1} is added to Q we still have $\mathsf{dist}[v_i] \le \mathsf{dist}[v_{i+1}]$. On the other hand, if $(v_i, v_{i+1}) \in E$, then after the for-loop $\mathsf{dist}[v_{i+1}]$ can only "go down" to $\mathsf{dist}[v_i] + c(v_i, v_{i+1})$. Since the costs are ***non-negative***, this is also $\ge \mathsf{dist}[v_i]$. Thus, at this point $\mathsf{dist}[v_{i+1}] \ge \mathsf{dist}[v_i]$. Since dist labels are not modified once they are set, we get a contradiction. □

Lemma 8 (Shortest Path Lemma)**.** Let $p = (s = x_0, x_1, \ldots, x_k)$ be any path. Let p_i be the sub-path $(x_0, \ldots, x_i)$. Then, $\mathsf{dist}[x_i] \le c(p_i)$, the cost of the path p_i, that is $c(p_i) = \sum_{j=0}^{i-1} c(x_j, x_{j-1})$.

Proof. Suppose not and choose the smallest i for which $\mathsf{dist}[x_i] > c(p_i)$. So $\mathsf{dist}[x_{i-1}] \le c(p_{i-1})$. Note that $c(p_{i-1}) = c(p_i) - c(x_{i-1}, x_i) < c(p_i)$ since the costs are ***non-negative***. Thus, $\mathsf{dist}[x_{i-1}] < \mathsf{dist}[x_i]$. Therefore, by Lemma 7, we get that x_{i-1} enters R before x_i. But then the for-loop for x_{i-1} would set $\mathsf{dist}[x_i] \le \mathsf{dist}[x_{i-1}] + c(x_{i-1}, x_i) \le c(p_{i-1}) + c(x_{i-1}, x_i) = c(p)$. Contradiction. □

Lemma 9. DIJKSTRA algorithm can be implemented in $O(m + n \log n)$ time.

Proof. This is a poster child application of priority queues. The data we wish to store as key-value pairs have keys given by the vertex identities and value of vertex v is $\mathsf{dist}[v]$. Line 14 is the EXTRACT-MIN operation. Line 12 is a DECREASE-VAL operation. The number of times Line 14 is implemented is at most $n + 1$ since the while loop runs for at most $n + 1$ times. The number of times Line 12 is implemented is at most the number of edges; each edge (v, u) appears when v enters Q and this happens at most once.

If we use the usual array implementation (see the file `graph_basics.pdf`), then the running time is $O(m + n^2)$. If we use the heap implementation, then the running time is $O((m + n) \log n)$. Using Fibonacci heaps, we get the $O(m + n \log n)$ running time. □

Theorem 2. In graphs with non-negative edge costs, DIJKSTRA algorithm can find the shortest paths from s to *every* vertex v in $O(m + n \log n)$ time.

The Shortest Path Tree. Just like in BFS(G, s), DIJSKTRA(G, s) also returns a tree rooted at s defined by the parents. More precisely, consider the graph on all vertices v with $\mathsf{dist}[v] < \infty$ where we add the edges $(\mathsf{parent}(v), v)$. This tree is called the *shortest path tree*.

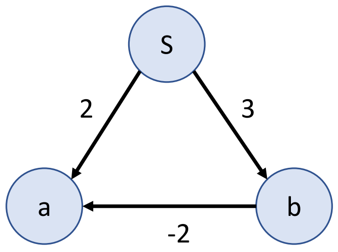

Figure 1: DIJKSTRA fails on this graph with negative costs

DIJKSTRA **doesn't work with negative cost edges.** Proofs to both lemmas, Lemma 7 and Lemma 8, crucially use the fact that the costs are non-negative. Indeed, they are otherwise false and in fact the algorithm fails even with one negative edge. Figure 1 shows an example.

First, note that the shortest cost path from s to a is actually (s, b, a) of total cost 1. Let's see what DIJKSTRA does. First s will assign a distance label $\mathsf{dist}[a] = 2$ and $\mathsf{dist}[b] = 3$. Then, it will pick a into R since $\mathsf{dist}[a]$ was the smaller one. And then, by design, it will *never* update $\mathsf{dist}[a]$ ever again. Remember, the reason DIJKSTRA did it to make sure the algorithm proceeds fast. However, the negative edge (b, a) is never allowed to update $\mathsf{dist}[a]$ again. And thus DIJKSTRA misses out and gives the wrong answer. Again, if there were no negative edges, this won't happen (as we just proved above).

It is very instructive to see that WBFS still works correctly on this example. s updates $\mathsf{dist}[a]$ to 2 and $\mathsf{dist}[b]$ to 3, and puts both in Q. Then a is popped/removed, and it doesn't update anything (there is no edge coming out of a). But then we do get b again, and b would update $\mathsf{dist}[a]$ to 1 (and $\mathsf{parent}[a]$ to b). This makes a come into the Q again, but then it doesn't bring anyone new, and the algorithm terminates. And note that Lemma 3 works *even* with negative costs. The issue with WBFS is that it may take a looong time to terminate.