CS 31: Algorithms (Spring 2019): Lecture 13

Date: 7th May, 2019

Topic: Graph Algorithms 3: Breadth First Search, Dijkstra

Disclaimer: These notes have not gone through scrutiny and in all probability contain errors. Please notify errors on Piazza/by email 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 $v \in V$.

```
procedure BFS($G, s$):
    ▷ Returns a distance label to every vertex.
    ▷ Every vertex not $s$ also has a pointer parent to another vertex.
    $\mathsf{dist}[s] \leftarrow 0$; $\mathsf{dist}[v] \leftarrow \infty$ otherwise.
    $\mathsf{parent}[v] \leftarrow \perp$ for all $v$.
    Assign queue $Q$ initialized to $s$. ▷ FIFO Queue
    while $Q$ is not empty do:
        $v = Q.\mathsf{remove}()$.
        for all neighbors $u$ of $v$ do:
            if ($\mathsf{dist}[u] > \mathsf{dist}[v] + 1$) then:
                Set $\mathsf{dist}[u] = \mathsf{dist}[v] + 1$.
                $Q.\mathsf{add}(u)$.
                Set $\mathsf{parent}[u] = v$.
```

Lemma 1. At any point of time let the $Q = [u_1, \ldots, u_k]$. Then $\mathsf{dist}[u_1] \leq \mathsf{dist}[u_2] \leq \cdots \leq \mathsf{dist}[u_k] \leq \mathsf{dist}[u_1] + 1$.

Proof. The proof is by induction over time. In the beginning, $Q = [s]$ and the lemma is vacuously true. Otherwise, fix an instant of time and let $Q = [u_1, \ldots, u_k]$, and let the lemma be true. In the next instant, we pop u_1 from Q and add in vertices $\{v_1, \ldots, v_r\}$ which are neighbors of u_1 with $\mathsf{dist}[v_i] = \mathsf{dist}[u_1]{+}1$. The Q at the next instant is $[u_2, u_3, \ldots, u_k, v_1, \ldots, v_r]$, and we see $\mathsf{dist}[u_2] \leq \ldots \leq \mathsf{dist}[u_k]$, by induction, $\mathsf{dist}[u_k] \leq \mathsf{dist}[v_1] = \mathsf{dist}[v_2] = \cdots = \mathsf{dist}[v_r] = \mathsf{dist}[u_1] + 1$ by induction, and algorithm design. Finally, $\mathsf{dist}[v_r] = \mathsf{dist}[u_1] + 1 \leq \mathsf{dist}[u_2] + 1$. Therefore, the condition holds in the next instant as well. □

The above lemma has a bunch of very useful corollaries.

Corollary 1 (Monotonicity Lemma)**.** Let $s = v_0, v_1, \ldots, v_t$ be the order in which vertices enter Q. Then,

$$\mathsf{dist}[v_0] \leq \cdots \leq \mathsf{dist}[v_t]$$

Proof. Follows from repeatedly applying the monotonicity lemma. □

Corollary 2. A vertex v never enters the queue more than once.

Proof. Suppose v enters the Q first time due to some vertex u. At that instant, $\mathsf{dist}[v] = \mathsf{dist}[u] + 1$. Henceforth, due to the above lemma, every vertex x that is popped from the Q has $\mathsf{dist}[x] \geq \mathsf{dist}[u]$, and in particular, we *never* have $\mathsf{dist}[v] > \mathsf{dist}[x] + 1$. Therefore, v never enters the Q again. □

Corollary 3 (BFS runtime)**.** BFS(G, s) runs in $\Theta(n + m)$ time.

Proof. Every edge (x, y) is scanned only when x removed from the Q. Since x only enters Q once, and therefore removed only once, each edge is scanned only once. □

Corollary 4. At the end, $\mathsf{dist}[v] = \mathsf{dist}[\mathsf{parent}[v]] + 1$ or ∞.

Proof. Since v only enters Q at most once, its distance label is set at most once. □

Now comes the main lemma which connects shortest paths and BFS¿

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

Before we prove the above statement, note that it implies the following as a corollary.

Lemma 3. Let $\mathsf{dist}[v]$ be the distance labels returned by BFS(G, s). Then $\mathsf{dist}[v]$ is the length of the shortest path from s to v.

Proof. Lemma 2 implies that $\mathsf{dist}[v]$ is at most the length of any path from s to v. Now that there is a path from s to v of length $\mathsf{dist}[v]$. This path is given by the reverse of $(v, \mathsf{parent}[v], \mathsf{parent}[\mathsf{parent}[v]], \ldots, s)$. □

Proof of Lemma 2. Suppose not, and let i be the smallest index for which this is false (note it is true for $i = 0$.) Thus, $\mathsf{dist}[x_{i-1}] \leq (i - 1)$ but $\mathsf{dist}[x_i] > i$. Now consider the time x_{i-1} is removed from the Q. At this point, the for-loop will check if $\mathsf{dist}[x_i] > \mathsf{dist}[x_{i-1}] + 1 \geq i$, and since this is true, it will set $\mathsf{dist}[x_i] \leq i$. Contradiction. □

Theorem 1. The shortest length path in an unweighted graph can be found in $\Theta(n + m)$ time.

The Shortest Path Tree. Note that 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. ✍

Exercise: *For practice, formally prove that T is a tree.*

Next, we wish to generalize to weighted graphs. It is indeed inviting to think of the following generalization for weighted graphs. The changes are marked in red.

```
procedure BFS($G, s$):
    ▷ Returns a distance label to every vertex.
    ▷ Every vertex not $s$ also has a pointer parent to another vertex.
    $\mathsf{dist}[s] = 0$; $\mathsf{dist}[v] = \infty$ otherwise.
    $\mathsf{parent}[v] = \perp$ for all $v \neq s$.
    Assign queue $Q$ initialized to $s$. ▷ FIFO Queue
    while $Q$ is not empty do:
        $v = Q.\mathsf{remove}()$.
        for all neighbors $u$ of $v$ do:
            if ($\mathsf{dist}[u] > \mathsf{dist}[v] + c(v, u)$) then:
                Set $\mathsf{dist}[u] = \mathsf{dist}[v] + c(v, u)$.
                $Q.\mathsf{add}(u)$.
                Set $\mathsf{parent}[u] = v$.
```

✍

Exercise: *Prove that the above algorithm is indeed correct. Show an example where the same vertex can enter the queue at least $\Omega(n)$ times. Show an example where the above algorithm may take $\Omega(mn)$ time. What is the best upper bound you can prove?*

2 Dijkstra's Algorithm

In this section, how we can make the algorithm described above "efficient" by not putting every eligible neighbor of the vertex in consideration in the queue. The algorithm works correctly when all costs are *non-negative*. This is the famous DIJKSTRA's algorithm named after its inventor, Edsger Dijkstra.

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 $v \in V$.

```
1:  procedure DISJKSTRA($G, s$):
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.
7:      while $Q$ is not empty do:
8:          $v \leftarrow Q$.remove()
9:          $R \leftarrow R + v$
10:         for all neighbors $u$ of $v$ do:
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:         $Q$.add$(u)$.
```

Remark: *The algorithm can also be seen as a generalization of the normal BFS. Think of the case when $c(e)$'s are positive intergers. One way to envisage the algorithm is by running BFS on a "bloated" graph where a graph with cost $c(e) = k$ is replaced by a path of length k between its endpoints. Once you do that, and focus only on the way the original vertices of the graph are picked in the queue, you see that it corresponds to Dijkstra;s algorithm.*

Lemma 4 (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 5 (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,

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

where 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, dist$[v_{i+1}] <$ 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 dist$[v_i] \leq$ 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, dist$[v_{i+1}]$ is not modified in that for-loop. Thus when v_{i+1} is added to Q we still have dist$[v_i] \leq$ 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 *positive*, this is also $\leq \mathsf{dist}[v_i]$. □

Lemma 6. Once a vertex x enters R its distance label dist and parent is never changed again.

Proof. Fix any vertex x and consider any while loop after x enters R. By the monotonicity property, when vertex u is selected in Line 14, $\mathsf{dist}[u] \geq \mathsf{dist}[x]$. Now, the for-loop of u would change x only if (a) (u, x) is a neighbor, and (b) $\mathsf{dist}[x] > \mathsf{dist}[u] + c(u, x)$. However, since *the costs are non-negative*, this would imply $\mathsf{dist}[x] > \mathsf{dist}[u]$ contradicting the property. Therefore, x's distance and therefore parent is never changed. □

As a corollary to the above, we get

Lemma 7. For any vertex x, $\mathsf{dist}[x] = \mathsf{dist}[\mathsf{parent}[x]] + c(\mathsf{parent}[x], x)$.

Proof. Look at the last time x's distance is set; this is when $\mathsf{parent}[x]$ enters R. At that point, as per the code, $\mathsf{dist}[x] = \mathsf{dist}[\mathsf{parent}[x]] + c(\mathsf{parent}[x], x)$. After this, by the above lemma $\mathsf{dist}[\mathsf{parent}[x]]$ remains unchanged, and $\mathsf{dist}[x]$ remains unchanged by definition of the last time. □

This implies the following: the reverse of the path $(x, \mathsf{parent}[x], \mathsf{parent}[\mathsf{parent}[x]], \cdots, s)$ is a path of cost precisely $\mathsf{dist}[x]$. ✍

Exercise: *Why is the above a path? Why can't there be any cycles? Hint: the "new" edges go from a vertex in R to a vertex outside R.*

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, $\mathsf{dist}[x_i] \leq c(p_i)$, the cost of the path p_i.

Proof of Lemma 8. Suppose not and choose the smallest i for which $\mathsf{dist}[x_i] > c(p_i)$. So $\mathsf{dist}[x_{i-1}] \leq c(p_{i-1})$. Note that $c(p_{i-1}) = c(p_i) - c(x_{i-1}, x_i) < c(p_i)$ *since the costs are positive*. Therefore, by the monotonicity Lemma 5, we get that x_{i-1} enters R before x_i. But then the for-loop for x_{i-1} would set $\mathsf{dist}[x_i] \leq \mathsf{dist}[x_{i-1}] + c(x_{i-1}, x_i) \leq c(p_{i-1}) + c(x_{i-1}, x_i) = c(p)$. □

We therefore get the following lemma.

Lemma 9. The $\mathsf{dist}[v]$ returned by Dijkstra are the costs of the minimum cost path from s to v in G. Furthermore, the path from s to v in the shortest path tree is the shortest path.

Proofs to both the lemmas above crucially uses the fact that the costs are postive. Indeed, they are otherwise false and in fact the algorithm fails. ✍

Exercise: *Come up with an example of a graph with one negative edge on which* DIJKSTRA *fails.*

Lemma 10. 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 $n+!$ 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 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 positive edge costs, DIJKSTRA algorithm can find the shortest paths from s to *every* vertex v in $O(m + n \log n)$ time.