Graphs : Shortest Paths : BFS + Dijkstra[1]

In the next few lectures we will look at an important suite of algorithms : finding shortest paths in graphs. The setting is of a *directed* graph $G = (V, E)$. Each edge e would have an associated cost[2] $c(e)$. The first two lectures will involve algorithms you have already seen earlier (CS 10, or sometimes even CS 1). These are the BFS and Dijkstra's algorithm. Our presentation will be slightly different, and will also *prove* why these algorithms are correct. Let us begin by stating the problem.

SINGLE SOURCE SHORTEST PATHS (SSSP)
Input: Directed Graph $G = (V, E)$, a source vertex $s \in V$, costs $c(e)$ on edges.
Output: Paths from s to every vertex $v \in V$ which have the smallest total cost.

Before we begin our algorithms, let us pause a moment and talk a bit about *certificates*. Any shortest path algorithm must also be able to *prove* that the paths returned are indeed the shortest. How would we prove something is the shortest? Indeed, this is not something specific to shortest paths; it is a universal question about any algorithm. How does an algorithm prove its correctness? What is the *certificate* that it has worked correctly. For example, in the knapsack problem, what is the certificate that the subset returned by the dynamic programming algorithm is the maximum cost one? The certificate is the table that it returns. If you go back and look at every algorithm we have seen so far, there are certificates that the algorithm also constructs on the way. Indeed, as problems become more complex, asking about these certificates often leads to good algorithms.

Back to shortest paths. How do we prove that a path p from s to v is the shortest? All the algorithms we will see in this class will use what are called ***distance labels*** as the certificates. A distance label $\mathsf{dist}(v)$ prescribes a value to every vertex v. Think of these labels as being *upper bounds* on the shortest path from s to v; we will always maintain the cost of the shortest path from s to v with be always $\leq \mathsf{dist}[v]$. We do this by making sure there is *some* path from s to v of that cost. What makes these distance labels *useful* is when they have the following property:

Definition 1 (Valid Distance Labels)**.** A distance label is an assignment $\mathsf{dist}(v)$ on *every* vertex of the graph G. A distance label dist is *valid* with respect to (G, s, c) if it satisfies the following

$$\mathsf{dist}(s) = 0, \quad \text{and for all edges } (u, v),\ \mathsf{dist}(v) \leq \mathsf{dist}(u) + c(u, v) \tag{1}$$

Theorem 1. Suppose d is a valid distance label w.r.t (G, s, c). Then the cost of *any* path from s to v *must* be ***at least*** $\mathsf{dist}(v)$.

Before we prove this above theorem, let us see why it gives certificates. For any v, we have a distance label $\mathsf{dist}[v]$ equaling the cost of some path from s to v. And the theorem says all other paths of cost at least that. And thus, we know that $\mathsf{dist}[v]$ is the shortest path length. Do such labels always exist? Can we find them? These are questions that should come to your mind, and indeed that is what we will see in the coming lectures. For now, let us prove the above theorem.

[1] *Lecture notes by Deeparnab Chakrabarty. Last modified : 19th Mar, 2022*
These have not gone through scrutiny and may contain errors. If you find any, or have any other comments, please email me at deeparnab@dartmouth.edu. Highly appreciated!

[2] These costs can be negative. For the first two lectures, we will assume that is ***not*** the case. In the last lecture of this week, we will also allow negative edges. As we will see, making costs negative will change the texture of the problem.

Proof of Theorem 1. Suppose not. Suppose there is a vertex v such that there is a path from s to v of cost $< \mathsf{dist}(v)$. Let this path be $(s =: x_0, x_1, x_2, \ldots, x_k := v)$. Let p_i denote the path from x_0 to x_i; so the path $p = p_k$. By our assumption, we have $c(p_k) < \mathsf{dist}(x_k)$. We also have $c(p_0) = \mathsf{dist}(x_0)$; indeed, p_0 is just a path with zero edges and $\mathsf{dist}(x_0) = \mathsf{dist}(s) = 0$ since d is valid. Let j be the *first* (smallest) index such that $c(p_j) < \mathsf{dist}(x_j)$. Such a j must exist and must be ≥ 1 and $\leq k$. By definition, $c(p_{j-1}) \geq \mathsf{dist}(x_{j-1})$ (otherwise, $j-1$ would have been the chosen index).

However, p_j is nothing but the path $(p_{j-1} \circ (x_{j-1}, x_j))$. Thus, $c(p_j) = c(p_{j-1}) + c(x_{j-1}, x_j) \geq \mathsf{dist}(x_{j-1}) + c(x_{j-1}, x_j)$. Since d is valid, $c(x_{j-1}, x_j) + \mathsf{dist}(x_{j-1}) \geq \mathsf{dist}(x_j)$. Therefore, we get $c(p_j) \geq \mathsf{dist}(x_j)$ contradicting our assumption that $c(p_j) < \mathsf{dist}(x_j)$. □

Remark: *Do valid distance labels always exist? The answer is no. If the graph G has a cycle C such that $\sum_{e \in C} c(e) < 0$, then valid distance labels cannot exist. For instance, suppose the cycle is (x_1, x_2, x_3, x_1). If valid distances labels did exist, then we would have $\mathsf{dist}(x_2) \leq \mathsf{dist}(x_1) + c(x_1, x_2)$, or, $c(x_1, x_2) \geq \mathsf{dist}(x_2) - \mathsf{dist}(x_1)$. Similarly, we would get $c(x_2, x_3) \geq \mathsf{dist}(x_3) - \mathsf{dist}(x_2)$ and $c(x_3, x_1) \geq \mathsf{dist}(x_1) - \mathsf{dist}(x_3)$. Adding all of these we would get $\sum_{e \in C} c(e) \geq 0$ which contradicts that the cycle has negative cost. Indeed, no one knows any "small" certificates for shortest paths in graphs which have negative cost cycles!*

1 Breadth First Search

We now begin algorithms for shortest paths starting with the easiest case when all $c(e) = 1$ for $e \in E$. In this case, the algorithm is BFS. All algorithms, however, follow a similar structure: maintain distance labels, fix violations to validity, and end when everything is valid. In BFS the algorithm scans through vertices in a queue fashion starting with neighbors of s, and then moving on to neighbors-of-neighbors, and so on.

```
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:      dist[s] ← 0; dist[v] ← ∞ otherwise.
5:      parent[v] ← ⊥ for all v.
6:      Assign queue Q initialized to s. ▷ FIFO Queue
7:      while Q is not empty do:
8:          v = Q.remove(). ▷ The first entry in Q
9:          for all neighbors u of v do: ▷ Update (v, u).
10:             if (dist[u] > dist[v] + 1) then:  ▷ dist[u] too large.
11:                 ▷ For BFS, in fact, dist[u] = ∞.
12:                 Set dist[u] = dist[v] + 1. ▷ Update dist[u]
13:                 Q.add(u). ▷ Since u's distance label was modified, put it in the Q
14:                 Set parent[u] = v.
```

We begin with a simple but key observation.

Observation 1. $\mathsf{dist}[v]$ of any vertex can never increase over time. Also, whenever $\mathsf{dist}[v]$ is modified, $\mathsf{dist}[v] = \mathsf{dist}[\mathsf{parent}[v]] + 1$. In general, $\mathsf{dist}[v] - \mathsf{dist}[\mathsf{parent}[v]] \geq 1$. Let T be the collection of edges $(\mathsf{parent}[v], v)$ at any point of the algorithm. This has no cycles and is thus a tree rooted at s.

Proof. We modify $\mathsf{dist}[u]$ to $\mathsf{dist}[v] + 1$ only if it was bigger than $\mathsf{dist}[v] + 1$. The also part follows from Line 12 and Line 14. When $\mathsf{dist}[v]$ is modified, we have $\mathsf{dist}[v] - \mathsf{dist}[\mathsf{parent}[v]] = 1$. At other times when $\mathsf{dist}[v]$ is unchanged but perhaps $\mathsf{dist}[\mathsf{parent}[v]]$ may go down (it doesn't for BFS), the difference can only go up. Suppose we do get a cycle $(x_1, x_2, \ldots, x_k, x_1)$. From the previous observation, $\mathsf{dist}[x_i] - \mathsf{dist}[x_{i-1}] \geq 1$ for all i, and adding this for all vertices in the cycle we get $0 \geq k \geq 1$, which is a contradiction. □

Lemma 1. If the BFS algorithm terminates, then it returns a valid distance label.

Proof. Since we assume BFS terminates, it ends with $\mathsf{dist}[v]$ on every vertex. Suppose, for the sake of contradiction, $\mathsf{dist}[v] > \mathsf{dist}[u] + 1$ for some (u, v) (recall, $c(u, v) = 1$). Since $\mathsf{dist}[u]$ is finite, it enters the Q at some time, and consider the last time $\mathsf{dist}[u]$ is being processed. At that time u is added to the Q, and consider when u reaches the front of the queue. At that for-loop, $\mathsf{dist}[v]$ is either $\leq \mathsf{dist}[u] + 1$ or set to $\mathsf{dist}[u]+1$. Subsequently, by Observation 1, $\mathsf{dist}[v]$ can only go down contradicting $\mathsf{dist}[v] > \mathsf{dist}[u]+1$. □

Remark: *There was absolutely nothing special about costs being* 1 *or even being positive in Observation 1 and Lemma 1. In particular, we can think of a* weighted BFS *where the only differences are in Line 10 and Line 12 where the* 1 *is changed to* $c(v, u)$*. The statements and proofs go through. However, the* ***if*** *in the latter's statement is a big if. Can you come up with an example of a graph with costs (negative allowed) where the weighted BFS would never terminate?*

We now show that when costs are unit, the BFS algorithm does terminate. In fact, the same vertex does not enter the queue more than once. The following *monotonicity lemma* is key.

Lemma 2. (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 $\mathsf{dist}[u_1] \leq \mathsf{dist}[u_2] \leq \cdots \leq \mathsf{dist}[u_k] \leq \mathsf{dist}[u_1] + 1$.

Before we prove the lemma, let us see why it is useful for analyzing the running time of BFS.

Claim 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 $\mathsf{dist}[x] \geq \mathsf{dist}[u]$. Contradiction. Therefore, no vertex enters the Q twice. □

Theorem 2. 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. Claim 1 shows that the algorithm terminates in $O(n + m)$ time since every vertex v enters once and takes $O(1 + \deg^+(v))$ time in the corresponding while loop. Summing over all vertices gives the runtime. Lemma 1 shows that $\mathsf{dist}[v]$'s are valid distance labels, and thus by Theorem 1, $\mathsf{dist}[v]$ 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. This is precisely the *reverse* of $(v, \mathsf{parent}[v], \mathsf{parent}[\mathsf{parent}[v]], \ldots, s)$. To see why, note that dist valid implies $\mathsf{dist}[v] - \mathsf{dist}[\mathsf{parent}[v]] \leq 1$ and Observation 1 implies the difference is ≥ 1. We must have equality. Indeed, the tree T defined in Observation 1 at end of BFS is called the ***shortest path tree***. □

Proof of Lemma 2. 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 $\mathsf{dist}[u_2] \leq \ldots \leq \mathsf{dist}[u_k]$. Furthermore, since $\mathsf{dist}[u_k] \leq \mathsf{dist}[u_1] + 1$ (by induction), we see $\mathsf{dist}[u_k] \leq \mathsf{dist}[x_i]$ for all $1 \leq i \leq r$. Finally, $\mathsf{dist}[x_i] = \mathsf{dist}[u_1] + 1 \leq \mathsf{dist}[u_2] + 1$. And thus, we get $\mathsf{dist}[u_2] \leq \ldots \leq \mathsf{dist}[u_k] \leq \mathsf{dist}[x_1] \leq \ldots \leq \mathsf{dist}[x_r] \leq \mathsf{dist}[u_2] + 1$. □

2 Dijkstra's Algorithm

In this section, we look at the famous generalization of BFS when costs can be arbitrary positive numbers. The difference with BFS is that it does not maintain a queue; rather it maintains dist[] labels initialized with $\mathsf{dist}[s] = 0$ and everything else ∞. Henceforth, it picks the vertex with the smallest dist and then *fixes* this label. For every unfixed neighbor, it updates the distance if that vertices distance label violates the validity of the distance label. Unlike BFS, it is immediate that the algorithm terminate, but it is not immediate it does so with valid distance labels. Indeed, if costs are negative, it does not.

```
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] = ∞ otherwise.
4:      parent[v] = ⊥ for all v ≠ s.
5:      Initialize R = ∅. ▷ R will be the "reached" vertices whose dist[v]'s never change.
6:      while R ≠ V do:
7:          Let v be the vertex ∉ R with smallest dist[v].
8:          if dist[v] = ∞ then:▷ No more vertices reachable from s
9:              Break ▷ Terminate
10:         R ← R + v
11:         for all neighbors u of v not in R do:  ▷ The distance labels set for only vertices outside R
12:             if (dist[u] > dist[v] + c(v, u)) then:
13:                 Set dist[u] ← dist[v] + c(v, u).
14:                 Set parent[u] ← v.
```

Lemma 3 (Termination)**.** The While loop in DIJKSTRA runs for at most n iterations.

Proof. At each while-loop iteration, a vertex is added into R and the algorithm stops when all vertices have been added. □

Lemma 4. Once a vertex x enters R its distance label dist and parent is never changed again. Furthermore, $\mathsf{dist}[x] = \mathsf{dist}[\mathsf{parent}[x]] + c(\mathsf{parent}[x], x)$.

Proof. This is made sure by Line 11 since it updates distances, if at all, only for vertices not in R. □

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, $\mathsf{dist}[v_{i+1}] < \mathsf{dist}[v_i]$. Since v_i was added to R before v_{i+1}, Line 7 implies the time when v_i is added to R, we have $\mathsf{dist}[v_i] \leq \mathsf{dist}[v_{i+1}]$. 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 R, $\mathsf{dist}[v_{i+1}]$ is not modified in that for-loop. Thus when v_{i+1} is added to R we still have $\mathsf{dist}[v_i] \leq \mathsf{dist}[v_{i+1}]$. 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 $\geq \mathsf{dist}[v_i]$. Thus, at this point $\mathsf{dist}[v_{i+1}] \geq \mathsf{dist}[v_i]$. Since dist labels are not modified once they are set, we get a contradiction. □

Lemma 6. At the end of Dijkstra's algorithm, the distance labels dist are valid.

Proof. Let (x, y) be any edge. If $\mathsf{dist}[y] \leq \mathsf{dist}[x]$, then since costs are ***non-negative***, $\mathsf{dist}[y] \leq \mathsf{dist}[x] + c(x, y)$ as well. If $\mathsf{dist}[y] > \mathsf{dist}[x]$, then by Lemma 5 x enters R before y, and then either $\mathsf{dist}[y] \leq \mathsf{dist}[x] + c(x, y)$ or is set to the latter. Since dist[] never increase, at the end we have $\mathsf{dist}[y] \leq \mathsf{dist}[x] + c(x, y)$. That is, the dist[] labels are valid. □

As in BFS, this proves that DIJKSTRA returns the shortest path from s to every vertex v: $\mathsf{dist}[v]$ is a lower bound on the shortest path, and the path which is the reverse of $(v, \mathsf{parent}[v], \mathsf{parent}[\mathsf{parent}[v]], \ldots, s)$ is indeed of this cost.

Lemma 7. 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 7 is the EXTRACT-MIN operation. Line 13 is a DECREASE-VAL operation. The number of times Line 7 is implemented is at most $n + 1$ since the while loop runs for at most $n + 1$ times. The number of times Line 13 is implemented is at most the number of edges; each edge (v, u) appears when v enters R 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 3. 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*.

DIJKSTRA **doesn't work with negative cost edges.** Proofs to both lemmas, Lemma 5 and Lemma 6, 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.

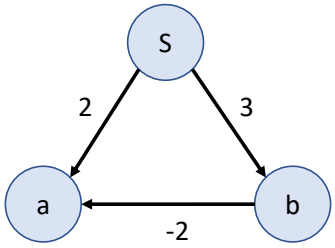

Figure 1: DIJKSTRA fails on this graph with negative costs

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 does this is 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).