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] $c(e)$. 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? First, note that *any* path from s to v provides an *upper bound* on the cost of the shortest path by definition. But even if we happen to chance upon a shortest path of a certain cost, how do we prove that there is nothing better? Is there a way to provide a *lower bound* to the cost of the shortest path? This is precisely what **distance labels** do.

Definition 1 (Valid Distance Labels)**.** *Let $G = (V, E)$ be a graph and $s \in V$ be a source vertex, and let $c(e)$ be costs on edges which could be positive or negative. A* distance label *is an assignment* $\mathsf{dist} : V \to \mathbb{R} \cup \{\infty\}$ *of a real number or "∞"[3] on every vertex of V. Such an assignment* dist *is* valid *with respect to (G, s, c) if it satisfies the following*

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

*The above inequality (1) vacuously holds true if $\mathsf{dist}(u) = \infty$ (even when $\mathsf{dist}(v) = \infty$) and **does not** hold if $\mathsf{dist}(v) = \infty$ but $\mathsf{dist}(u)$ is finite.*

Definition 2 (Tight Edges)**.**
Let $d : V \to \mathbb{R} \cup \{\infty\}$ be a valid distance label with respect to (G, s, c). An edge (u, v) is called tight w.r.t d if the inequality in (1) holds with equality.

[1] *Lecture notes by Deeparnab Chakrabarty. Last modified : 26th Feb, 2025*
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.

[3] one should think of ∞ as "undefined" rather than a large value; see definition below

Theorem 1 (Valid Distance Labels are Lower Bounds)**.** Suppose dist 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*** $\mathsf{dist}(v)$. In particular, if $\mathsf{dist}(v) = \infty$, then this must mean there is *no* path from s to v in G since every path has finite cost.

Proof. Fix a vertex v and fix a path $p = (s = x_0, x_1, \ldots, x_k := v)$. We will use the properties of valid distance labels to prove $c(p) \geq \mathsf{dist}(v)$. Indeed, we know

$$\forall 1 \leq i \leq k, \;\; \mathsf{dist}(x_i) \leq \mathsf{dist}(x_{i-1}) + c(x_{i-1}, x_i)$$

Adding this all up, and cancelling $\mathsf{dist}(x_1)$ to $\mathsf{dist}(x_{k-1})$ from LHS and RHS, we get

$$\mathsf{dist}(x_k) \leq \underbrace{\mathsf{dist}(x_0)}_{=\mathsf{dist}(s)=0} + \underbrace{\sum_{i=1}^{k} c(x_{i-1}, x_i)}_{=c(p)} = c(p)$$ □

Theorem 2 (Tight Edges and Shortest Paths)**.** Suppose dist be a valid distance label w.r.t (G, s, c). If there is a path p from s to a vertex v such that all edges in the path are *tight*, then p must be a shortest cost path from s to v.

Proof. Let $(s = x_0, x_1, \ldots, x_k = v)$ be the path. We have (x_i, x_{i+1}) is tight for $0 \leq i \leq k-1$. That is,

$$\mathsf{dist}[x_{i+1}] = \mathsf{dist}[x_i] + c(x_i, x_{i+1}), \;\; \forall 0 \leq i \leq k-1$$

Adding all of these up gives,

$$\sum_{i=0}^{k-1} \mathsf{dist}[x_{i+1}] = \sum_{i=0}^{k-1} \mathsf{dist}[x_i] + c(p)$$

where $c(p)$ is the cost of the path, that is, the sum of costs of the edges. Rearranging, and noting $\mathsf{dist}[x_0] = \mathsf{dist}[s] = 0$ and $\mathsf{dist}[x_k] = \mathsf{dist}[v]$, we get $\mathsf{dist}[v] = c(p)$. However, by Theorem 1, we know $c(q) \geq \mathsf{dist}[v]$ for any path q from s to v. Therefore, p is a shortest cost path from s to v. □

Theorem 3 (Tight Edges are Acyclic)**.** Suppose dist be a valid distance label w.r.t (G, s, c). Let F be the collection of all *tight* edges. Then, if all cycles in G have *positive* total cost, then $H = (V, F)$ is acyclic.

Proof. If there is a cycle $(x_1, \ldots, x_k, x_1)$ in $H = (V, F)$, then since every edge is tight, we get

$$\mathsf{dist}[x_{i+1}] - \mathsf{dist}[x_i] = c(x_i, x_{i+1}), \; \forall 1 \leq i \leq k-1, \;\; \text{and} \;\; \mathsf{dist}[x_1] - \mathsf{dist}[x_k] = c(x_k, x_1)$$

Adding all of these we see that the sum of the LHS-es telescopes to 0, while the RHS sums to the cost of the cycle. Which is positive. Contradiction. □

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.*

1 The Generic SSSP Algorithm

Armed with the definition of valid distance labels, and their utility as underscored by the previous theorems, we are now ready to describe the *generic* SSSP algorithm. By design some things will be left ill-defined. The idea is very simple. We begin with distance label 0 on s and ∞ every where else. At this point, the only edges which are "bad" (more precisely, edges (x, y) with $\mathsf{dist}(y) > \mathsf{dist}(x) + c(x, y)$) are the ones out-going from s. So, we add s to the set Q of "queasy" vertices; vertices, which *may* have out-going edges which are bad. Note, when the graph is undirected, these are just all incident edges. Henceforth, the algorithm proceeds in rounds. In each round, we pick a queasy vertex x (for now, arbitrarily), we "fix" all edges (x, y) by decreasing the distance label of y if it is too high, and in case we do so, we remember this using a "parent" data structure, and, more importantly, we add y to Q since decreasing y's distance label *may* make "good" edges of the form (y, z) turn bad — y is queasy, y gets into Q. Ultimately, if nothing is queasy, we should have all edges good. Voila! We have valid distance labels. (Don't worry, proofs are forthcoming). The detailed pseudo-code is as follows.

```
1:  procedure SSSP(G, s, c): ▷ Nothing is assumed about c
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. ▷ These distance labels are initially not valid.
5:      parent[v] ← ⊥ for all v.
6:      Q initialized with s. ▷ Think of Q as a set.
7:      ▷ Invariant: if (x, y) is "bad" wrt dist, then x will be in Q.
8:      while Q is not empty do:
9:          v ← Q.remove(). ▷ At this point, any which way.
10:         for all out-neighbors u of v do: ▷ Update (v, u).
11:             if (dist[u] > dist[v] + c(v, u)) then:  ▷ dist[u] too large
12:                 Set dist[u] ← dist[v] + c(v, u). ▷ Update label of u
13:                 Set parent[u] ← v. ▷ Update parent of u
14:                 Q.add(u). ▷ Since u's distance label was modified, put it in Q
```

A notion of "time". To argue about the correctness of this algorithm, it will be convenient to have a notion of time for the above algorithm. Informally, think of running this algorithm on an instance and being able to "pause" it any time and inspecting the values of the various variables. More formally, we can think of a variable t which increments by 1 whenever Line 11 to Line 14 runs (it increments even when the if-statement isn't true). We let $\mathsf{dist}_t[x]$ be the value of the variable $\mathsf{dist}[x]$ at the *end* of the tth run of those lines. We begin with simple but key observations.

Lemma 1. The following invariants hold true for SSSP.

a. For $t < t'$, $\mathsf{dist}_t[v] \geq \mathsf{dist}_{t'}[v]$.
b. If $\mathsf{dist}_t[v] \neq \mathsf{dist}_{t-1}[v]$, that is, if $\mathsf{dist}[v]$ is modified at time t, then $\mathsf{dist}_t[v] = \mathsf{dist}_t[\mathsf{parent}_t[v]] + c(\mathsf{parent}_t[v], v)$.
c. Furthermore at any time t such that $\mathsf{parent}_t[v] \neq \perp$, $\mathsf{dist}_t[v] \geq \mathsf{dist}_t[\mathsf{parent}_t[v]] + c(\mathsf{parent}_t[v], v)$.

Proof. Part (a) follows since we modify $\mathsf{dist}[u]$ to $\mathsf{dist}[v] + c(u, v)$ only if it was bigger than $\mathsf{dist}[v] + c(u, v)$. Part (b) follows from Line 12 and Line 13. To see part (c), consider two consecutive times t_1 and t_2 when $\mathsf{dist}[v]$ is modified. The inequality is true with equality for $t = t_1$ and $t = t_2$ from part (b). Now consider any time $t \in (t_1, t_2)$. The only way this equality is violated is if $\mathsf{dist}[\mathsf{parent}_t[v]]$ is modified. However, part (a) tells us that in that case it can only go down, which would make the inequality in the direction claimed. □

Theorem 4. If the SSSP algorithm terminates, then (a) it returns valid distance labels, (b) if $\mathsf{parent}[v] \neq \perp$, then $(\mathsf{parent}[v], v)$ edge is a *tight* edge.

Proof. Let's prove part (a) first. Since we assume SSSP terminates, it ends at some time T with $\mathsf{dist}[v] = \mathsf{dist}_T[v]$ on every vertex. Fix an edge (u, v). Suppose, for the sake of contradiction, $\mathsf{dist}[v] > \mathsf{dist}[u] + c(u, v)$ for some (u, v) (recall, $c(u, v) = 1$). Note that this $\mathsf{dist}[u]$ and $\mathsf{dist}[v]$ is at the end of the algorithm. Since $\mathsf{dist}[u]$ is finite (otherwise the inequality cannot hold), it has been set in Line 12 at some point of time. Let t be the last time when $\mathsf{dist}[u]$ was modified, and therefore by definition, for any time $\tau > t$ we have $\mathsf{dist}_\tau[u] = \mathsf{dist}[u]$. Since the distance was modified, due to Line 14, u was added to Q at time t. Since the algorithm terminates, there is some time $> t$ when u comes to the front of the queue and is removed. We then run the for-loop over out-neighbors of u and in one of those, at time t', u encounters v. At that time, t', we either have $\mathsf{dist}_{t'}[v] \leq \mathsf{dist}_{t'}[u] + c(u, v)$ or is set to $\mathsf{dist}_{t'}[u] + c(u, v)$. In either case, $\mathsf{dist}_{t'+1}[v] \leq \mathsf{dist}_{t'+1}[u] + c(u, v) = \mathsf{dist}[u] + c(u, v)$, where the last equality follows since $t' + 1 > t$. Finally, by part (a) of Lemma 1, we get $\mathsf{dist}[v] = \mathsf{dist}_T[v] \leq \mathsf{dist}_{t'+1}[v]$, and so, together, we get $\mathsf{dist}[v] \leq \mathsf{dist}[u] + c(u, v)$ proving that d satisfies all the distance label conditions.

To prove part (b), we simply invoke part (c) of Lemma 1. Since dist is a valid distance label, the inequality of part (c) must occur with equality. This implies all $(\mathsf{parent}[v], v)$ edges are tight. □

Corollary 1 (Shortest Path Tree.)**.** *The collection of* $(\mathsf{parent}[v], v)$ *edges form a* directed *out-tree from* s*, and the path from* s *to* v *in this tree is a shortest path from* s *to* v*. This tree is called the* shortest path tree.

Proof. Follows from part (b) of Theorem 4, Theorem 2, and Theorem 3. □

Remark: *Note that the* ***if*** *in the latter's statement is a big if; after all we know that distance labels don't exist if* s *can reach a negative cost cycle. Indeed, run the above algorithm on such a graph to get a feel of how it goes into an infinite loop.*

2 Breadth First Search: $c \equiv 1$

We now specialize SSSP in the case when all costs $c(e) = 1$. In this case, we maintain Q as a FIFO queue. That is, whenever we run Line 9, we remove from the "front" of the queue, and when we add in Line 14, we add to the back of the queue. We will argue that with this modification, when $c \equiv 1$, the algorithm SSSP, which is called *breadth first search* or BFS, will not only terminate but rather will do so in pretty fast, in particular, in $O(m + n)$ time. Just for completeness, we describe the whole algorithm again.

```
1:  procedure BFS(G, s, c ≡ 1): ▷ All c(e)'s are 1 (or equivalently, the same)
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. ▷ These distance labels are initially not valid.
5:      parent[v] ← ⊥ for all v.
6:      Q is initialized as a FIFO queue; Q.add(s).
7:      ▷ Invariant: if (x, y) is "bad" wrt dist, then x will be in Q.
8:      while Q is not empty do:
9:          v ← Q.remove(). ▷ v us first entry of Q
10:         for all out-neighbors u of v do: ▷ Update (v, u).
11:             if (dist[u] > dist[v] + 1) then:  ▷ Recall, all costs are 1.
12:                 Set dist[u] ← dist[v] + 1. ▷ Update label of u
13:                 Set parent[u] ← v. ▷ Update parent of u
14:                 Q.add(u). ▷ Since u's distance label was modified, put it in Q
```

The key lemma which will drive the running time argument is the following *monotonicity lemma*. This lemma **strongly** uses both $c \equiv 1$ and that Q is a FIFO queue.

Lemma 2. (Monotonicity Lemma for BFS.) In BFS, consider the vertices v removed in Line 9 at time t, and define $\delta_t := \mathsf{dist}_t[v]$. Then, δ_t's are non-decreasing over time. More precisely, if δ_t and $\delta_{t'}$ are defined for two different while loops, $t < t'$ implies $\delta_t \leq \delta_{t'}$.

Proof. To prove this lemma, we will need a (stronger) claim about the state of the FIFO queue.

Claim 1. *Fix a time t when we are about to begin a while loop. Let $Q = [u_1, \ldots, u_k]$ be the queue content at this time. Then $\mathsf{dist}_t[u_1] \leq \mathsf{dist}_t[u_2] \leq \cdots \leq \mathsf{dist}_t[u_k] \leq \mathsf{dist}_t[u_1] + 1$.*

Proof of Claim 1. 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 $\mathsf{dist}_t[x] > \mathsf{dist}_t[u_1]+1$, and upon adding, the distances become $\mathsf{dist}_{t+1}[x] = \mathsf{dist}_t[u_1] + 1$. Let these neighbors be $x_1, \ldots, x_r$ (note r could be 0 and there could be no such neighbors). After u_1 is process and we are about to move to the next while loop, the time has ticked to $t' > t$ (it is precisely $t+$ out-degree of u_1). Note that the contents of the Q at t' is $[u_2, u_3, \ldots, u_k, x_1, \ldots, x_r]$. We now wish to prove the assertion in the claim for t'.

First, by induction, we know for $2 \leq i \leq k$, we have $\mathsf{dist}_t[u_i] \leq \mathsf{dist}_t[u_1] + 1$, and so the $x_1, \ldots, x_r$ are distinct from $u_2, \ldots, u_k$. And so, $\mathsf{dist}_{t'}[u_i] = \mathsf{dist}_t[u_i]$ for $2 \leq i \leq k$. And so it remains true that $\mathsf{dist}_{t'}[u_2] \leq \cdots \leq \mathsf{dist}_{t'}[u_k]$. Furthermore, since $\mathsf{dist}_t[u_k] \leq \mathsf{dist}_t[u_1]+1$ (by induction) and since $\mathsf{dist}_{t'}[x_j] = \mathsf{dist}_t[u_1]+$

1, we see $\mathsf{dist}_{t'}[u_k] \leq \mathsf{dist}_{t'}[x_i]$ for all $1 \leq i \leq r$. Finally, $\mathsf{dist}_{t'}[x_i] = \mathsf{dist}_t[u_1] + 1 \leq \mathsf{dist}_t[u_2] + 1$, where the inequality again follows by induction. And this completes the proof of the claim. □

The lemma almost immediately follows from the previous claim. Fix x which is removed at time t and $\delta_t = \mathsf{dist}_t[x]$. If at that point Q has another vertex y, then y would be the next removed vertex at some $t' > t$, and note the previous claim shows $\mathsf{dist}_{t'}[y] \geq \mathsf{dist}_t[x]$ and so $\delta_{t'} \geq \delta_t$. If Q had only x, then if any vertex y is added by x, then its will have $\mathsf{dist}_{t'}[y] = \mathsf{dist}_t[x] + 1$ and so $\delta_{t'} = \delta_t + 1$. □

Next, we use Lemma 2 to show no vertex enters Q more than once. It is instructive to note that the following claim will not need Q is a FIFO queue, and will only need that costs are positive.

Claim 2. *In the* BFS *algorithm, the same vertex v is never removed from the queue more than once. In other words, the vertices encountered in Line 9 of* BFS *are all distinct.*

Proof. Suppose not, and suppose v is removed from Q at time t_1 and at $t_2 > t_1$. When v is removed at time t_1, we have $\mathsf{dist}_{t_1}[v] = \delta_{t_1}$ (definition of δ_{t_1}). Since v is removed at t_1, this means there is some time between t_1 and t_2 when v is added to Q again. Suppose $x \in Q$ is responsible for this and say x was removed from Q at time $\tau \in (t_1, t_2)$. Then note that since x adds v into Q in this while loop, we must have $\mathsf{dist}_\tau[v] > \mathsf{dist}_\tau[x] + 1 = \delta_\tau + 1 > \delta_\tau$. Since $\tau > t_1$, by Lemma 1, we have $\mathsf{dist}_\tau[v] \leq \mathsf{dist}_{t_1}[v]$. Putting together, we get $\delta_{t_1} > \delta_\tau$ contradicting Lemma 2, since $t_1 < \tau$. □

Theorem 5. When $c \equiv 1$ and Q is implemented as a FIFO queue in BFS, one can find valid distance labels and the shortest paths a vertex s to every other reachable vertex v in $O(n + m)$ time.

Proof. Claim 2 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 run-time. Theorem 4 and Corollary 1 prove the theorem. □

3 Dijkstra's Algorithm: when costs are non-negative

In the previous section, we defined the algorithm BFS when $c \equiv 1$ which was the SSSP algorithm with Q implemented as a FIFO queue. In this section, we look at the specialization of SSSP to *positive* costs, and this will be Dijkstra's algorithm. Before we go there, let us ask why BFS itself (which was for $c \equiv 1$) doesn't quite work. In particular, if we just used a FIFO queue but costs were not all equal, we see an example where Claim 2 is violated; indeed, the reader should work out what BFS does on the example in Figure 1.

Remark: *Although Claim 2 is violated, can one still show that* BFS *on (G, s, c) when run with positive costs and Q as a FIFO queue will actually terminate in finite time? In particular, can you show that the same vertex can't be removed* infinitely *many times? Can you then prove an upper bound on this time as a function of m and n?*

Once you see that vertices may repeatedly be removed from Q, one notes that the "monotonicity lemma" (Lemma 2) must be false since that was key to proving Claim 2. The fix is to "force" the "monotonicity lemma" to almost hold by *design*: the way to do this is that when we remove a vertex from Q in Line 9, instead of choosing the "first" vertex, we choose the vertex with the *smallest* $\mathsf{dist}[x]$.

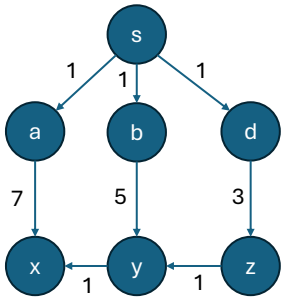

Figure 1: *Does Claim 2 hold if we run* $\text{BFS}(G, s, c)$ *above with marked costs?*

```
1:  procedure PositiveBFS(G, s, c): ▷ We assume costs are non-negative on every e ∈ E
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. ▷ These labels are initially not valid
5:      parent[v] ← ⊥ for all v.
6:      Q is initialized to s. ▷ Go back to thinking Q as a set
7:      while Q is not empty do:
8:          v ← min_{x∈Q} dist[x]; Q.remove(v). ▷ Implementation Details Later
9:          for all neighbors u of v do: ▷ Update (v, u).
10:             if (dist[u] > dist[v] + c(v, u)) then:  ▷ dist[u] too large
11:                 Set dist[u] ← dist[v] + c(v, u). ▷ Update dist[u]
12:                 Set parent[u] ← v.
13:                 Q.add(u). ▷ If u is already in Q then it isn't added again.
```

By Theorem 4 and Corollary 1, if the above algorithm terminates it'll give us valid distance labels and shortest paths. We first need to show that the above algorithm terminates. To that end, we will prove Claim 2 for the above modification. And to do so, we will prove analog of Lemma 2 for the above algorithm. We restate it here.

Lemma 3. (Monotonicity Lemma for PositiveBFS.) In PositiveBFS with non-negative costs, consider the vertices v removed in Line 8 at time t, and define $\delta_t := \text{dist}_t[v]$. Then, δ_t's are non-decreasing over time. More precisely, if δ_t and $\delta_{t'}$ are defined for two different while loops, $t < t'$ implies $\delta_t \leq \delta_{t'}$.

Proof. Let us consider two consecutive removals (i.e, two consecutive executions of Line 8) by the algorithm at time t_1 and t_2. Let these vertices be x and y, respectively. Note, by definition, we have $\delta_{t_1} = \text{dist}_{t_1}[x]$ and $\delta_{t_2} = \text{dist}_{t_2}[y]$. Therefore, if we show $\text{dist}_{t_1}[x] \leq \text{dist}_{t_2}[y]$, then we would be done.

Two cases arise. The for-loop of x, which runs from time t_1 to t_2 either modifies $\text{dist}[y]$ or it doesn't. If it doesn't, then y wasn't added to Q in x's for-loop. This means y was in Q at time t_1 and by choice of x (Line 8) we must have $\text{dist}_{t_1}[x] \leq \text{dist}_{t_1}[y] = \text{dist}_{t_2}[y]$, since y's distance label wasn't changed from t_1 to t_2. The other case is x's for-loop does modify $\text{dist}[y]$. However, this means $\text{dist}_{t_2}[y] = \text{dist}_{t_1}[x] + c(x, y)$. And *since costs are non-negative*, we get $\text{dist}_{t_2}[y] \geq \text{dist}_{t_1}[x]$ in this case as well. Thus, in either case, we have proved what we wanted. □

The proof of the following claim is literally almost the same as the proof of Claim 2.

Claim 3. *In the* POSITIVEBFS *algorithm, the same vertex v is never removed from the queue more than once. In other words, the vertices encountered in Line 8 of* POSITIVEBFS *are all distinct.*

Proof. Suppose not, and suppose v is removed from Q at time t_1 and at $t_2 > t_1$. When v is removed at time t_1, we have $\mathsf{dist}_{t_1}[v] = \delta_{t_1}$ (definition of δ_{t_1}). Since v is removed at t_1, this means there is some time between t_1 and t_2 when v is added to Q again. Suppose $x \in Q$ is responsible for this and say x was removed from Q at time $\tau \in (t_1, t_2)$. Then note that since x adds v into Q in this while loop, we must have $\mathsf{dist}_\tau[v] > \mathsf{dist}_\tau[x] + c(x,v) = \delta_\tau + c(x,v) \geq \delta_\tau$. Since $\tau > t_1$, by Lemma 1, we have $\mathsf{dist}_\tau[v] \leq \mathsf{dist}_{t_1}[v]$. Putting together, we get $\delta_{t_1} > \delta_\tau$ contradicting Lemma 3, since $t_1 < \tau$. □

The above lemma shows that POSITIVEBFS runs at most n while loops. How much time does each while-loop take? The for-loop Line 9 for x, as noted, takes $O(1 + \deg^+(x))$ time (as in BFS). However, how much time does Line 8 take? Naively, or rather if $\mathsf{dist}[\cdot]$ is kept as an array, then this can take $O(n)$ time (which dominates $1 + \deg^+(x)$). And if we do so, we get an $O(n^2)$ running time. Which is okay, if G was a dense graph. However, if G is sparse (with $m \ll n^2$), then a different *data-structure* is needed. Let's take a detour.

Priority Queues

Consider the following *data structure*. Here is the task. There is a set Q of objects with (key, value) pairs whose keys come from an n element universe U. We may as well assume the keys are numbers from 1 to n. We want to allow the following 4 operations:

- INSERT(Q, x): Insert an object x into S.
- DELETE(Q, x): Delete an object x from S. We won't need this for shortest paths.
- DECREASE-VAL(Q, x, v): Decrease the value of $x \in S$ to v only if v is smaller than x's current value.
- EXTRACT-MIN(Q): Return the $x \in Q$ with minimum value and delete it.

Of course one can just use an array $A[1:n]$ where $A[x]$ stores the value of $x \in S$ and $\perp$ otherwise. The first three operations take $O(1)$ time, however, the last operation takes $\Theta(n)$ time. On the other hand if we store the items as a MIN-HEAP, then *all* the operations take $O(\log n)$ time. Using heaps to implement priority queues is the most common way. There is another data structure called the FIBONACCI HEAP which can implement the first three operations in $O(1)$ time[4] and the last in $O(\log n)$ time. Seems like the best of both the array-and-the-heap world. The following table encapsulates all this.

Operation	Array	Heap	Fibonacci Heap
INSERT(Q, x)	$O(1)$	$O(\log n)$	$O(1)$
DELETE(Q, x)	$O(1)$	$O(\log n)$	$O(1)$
DECREASE-VAL(Q, x, v)	$O(1)$	$O(\log n)$	$O(1)$
EXTRACT-MIN(Q)	$O(n)$	$O(\log n)$	$O(\log n)$

Back to Shortest Paths

The above discussion hopefully tells you how to implement POSITIVEBFS, but we spell it out.

[4] I am lying a bit. The $O(1)$ is only *amortized* over many calls; more precisely, if t such calls are made, then they cost $O(t+n)$, and so it $t = \Theta(n)$, this amortizes to $O(1)$ per call.

- We initialize our priority queue Q with $(v, \mathsf{dist}[v])$ with n INSERTS. You can think of ∞ as sum of all the edge costs plus one.
- Line 8 is an EXTRACT-MIN(Q) operation.
- Line 11 is a DECREASE-VAL operation where we decrease the value of u to $(\mathsf{dist}[v] + c(v, u))$.

```
1:  procedure PositiveBFSwithPriorityQueues(G, s, c): ▷ Assume c ≥ 0.
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. ▷ These labels are initially not valid
5:      parent[v] ← ⊥ for all v.
6:      Initialize priority queue Q. INSERT(Q, (s, 0)). ▷ Key: vertex name. Value: Distance Label
7:      while Q is not empty do:
8:          v ← EXTRACT-MIN(Q).
9:          for all neighbors u of v do: ▷ Update (v, u).
10:             if (dist[u] > dist[v] + c(v, u)) then:
11:                 Set dist[u] ← dist[v] + c(v, u). ▷ Update dist[u]
12:                 Set parent[u] ← v.
13:                 ▷ Do the Priority Queue Management
14:                 if u ∈ Q then:
15:                     DECREASE-VAL(Q, u, dist[v] + c(v, u)). ▷ Update dist[u]
16:                 else:
17:                     INSERT(Q, (u, dist[u])) ▷ Add u to Q if it isn't already present
```

With the discussion from previous subsection, we get the following theorem on the running time of POSITIVEBFSWITHPRIORITYQUEUES.

Theorem 6. The algorithm POSITIVEBFSWITHPRIORITYQUEUES when Q is a priority queue implemented using Fibonacci Heaps solves the SSSP problem in $O(m + n \log n)$ time.

Proof. The proof of correctness has already been discussed above. The runtime is simply because DECREASE-VAL is an $O(1)$ time operation implying the for-loop for vertex x takes $O(1 + \deg^+(x))$ time, and Line 8 takes $O(\log n)$ time. Put all together, we get time bounded by $O\left(\sum_{x \in V} \log n + 1 + \deg^+(x)\right) = O(m + n \log n)$. □

"Usual" Dijkstra's Algorithm: a reinterpretation

The algorithm POSITIVEBFS is an algorithm initially posited by, among many other people, Edsger Dijkstra, and this algorithm has since been called Dijkstra's algorithm. With one *annoying* catch. If you look in (most) textbooks or the web, it is presented slightly differently which, as far as I can see, has almost no advantage. However, just to make you aware of this not-so-subtle difference, let me tell what the difference is. It stems from the observation that when all costs are non-negative, when a vertex v is removed from the "queue", it never changes its distance label. Indeed, this follows from Claim 3: if it had changed its distance label, then it would come back in Q and would then be removed from Q violating Claim 3. The "usual Dijkstra exposition" actually *hard-codes* this and this is how.

```
1:  procedure DIJKSTRA(G, s, c):▷ costs assumed to be positive
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:      ▷ One should think of R as "removed" vertices in POSITIVEBFS.
7:      while R ≠ V do:
8:          Let v be the vertex ∉ R with smallest dist[v].
9:          R ← R + v
10:         for all neighbors u of v do:  ▷ The distance labels set for only vertices outside R
11:             if (dist[u] > dist[v] + c(v, u)) then:
12:                 Set dist[u] ← dist[v] + c(v, u).
13:                 Set parent[u] ← v.
```

Lemma 4. When costs c are positive, then the output $\mathsf{dist}[v]$ and $\mathsf{parent}[v]$ of POSITIVEBFS is exactly the same as that of DIJKSTRA.

Proof. (Sketch). In POSITIVEBFS if we defined a set R which was the set of vertices removed in Line 8, then in every while loop we add one vertex to R, and Claim 3 shows that the same vertex is never added twice to R. Furthermore, the vertex added to R is the vertex in Q with the smallest $\mathsf{dist}[v]$; but this is also the one with smallest $\mathsf{dist}[v]$ in $V \setminus R$ because all vertices in $V \setminus (Q \cup R)$ have $\mathsf{dist}[v] = \infty$. Therefore, the behavior is precisely as in DIJKSTRA. □

Theorem 7. In graphs with positive edge costs, DIJKSTRA finds the shortest paths from s to *every* vertex v in $O(m + n \log n)$ time if the implementation is via Fibonacci heap priority queues.

When costs are positive, the above two algorithms are one and the same. It is, however, important to remember that if costs are negative, the above implementation DIJKSTRA could return the wrong answer. Here is a simple example on a DAG. On the other hand POSITIVEBFS will work just fine on the above example (I'll let you check this).

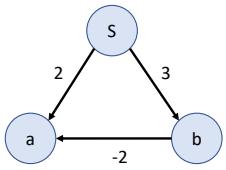

Figure 2: DIJKSTRA fails on this graph with negative costs. First, note that the shortest cost path from s to a is actually (s, b, a) of total cost 1. Let's see what DIJKSTRA does. First s will assign a distance label $\mathsf{dist}[a] = 2$ and $\mathsf{dist}[b] = 3$. Then, it will pick a into R since $\mathsf{dist}[a]$ was the smaller one. And then, by design, it will *never* update $\mathsf{dist}[a]$ ever again.

Once again, I am actually not 100% sure why various sources (textbooks/web-entries, etc) don't teach POSITIVEBFS instead of DIJKSTRA; the only exception I am aware of is Jeff Erickson and even he has the

"version" above (which we call DIJKSTRA) with the name NONNEGATIVEDIJKSTRA (See page 289 of his notes/book). Maybe I will call POSITIVEBFS just DIJKSTRA in my next iteration of these notes.

Remark: *Can you show whether* POSITIVEBFS *always terminate in finite time if G doesn't have negative cost cycles? Can you bound this time as a function of n and m?*