1 SSSP in Graphs with Nonnegative Weight Edges

We are again considering SSSP in weighted graphs, but today we will restrict the edge weights to be nonnegative:

Single Source Shortest Paths (SSSP) in nonnegatively weighted graphs: Given a directed or undirected $G = (V, E)$ weighted by $w : E \to \mathbb{Z}^+ \cup \{0\}$, and a source vertex $s \in V$, compute for every vertex v, the distance $d(s, v)$ and a predecessor vertex $\pi(v)$ so that $(\pi(v), v) \in E$ and $d(s, v) = d(s, \pi(v)) + w(\pi(v), v)$.

We already saw a fast algorithm for a special case of nonnegative weights SSSP: when the edge weights are all 1. This is the case of unweighted graphs. We solved SSSP in unweighted graphs in linear time using BFS.

Let us consider what BFS does. We said that for increasing values of the distance i from the source s, in each iteration i it considers the set of vertices L_i at distance i from s which were computed in the previous iteration $i-1$ (in iteration 0, $L_0 = \{s\}$). Then one builds up L_{i+1}, by going through each edge (u, v) where $u \in L_i$ and checking whether v was already visited, and if it is not, then we stick v into L_{i+1}.

An analogous way to view BFS is in the relaxation framework. The distance estimates $d[v]$ are set to ∞ for all v except for the source s for which $d[s]$ is set to 0. This was done implicitly before by setting all vertices to be unvisited except for s. Now, let us imagine that in iteration i of BFS we had a queue of vertices Q that contains vertices at distance i and $i+1$ from s so that the vertices of distance i are first. Then, we can view BFS as popping the first vertex u from Q, going through its edges (u, v) and relaxing them: i.e. if $d[v] > d[u] + 1$, then set $d[v] = d[u] + 1$ and place v at the end of Q. If $d[u] = i$, then this corresponds exactly to checking whether v has been visited before and if not, placing it into L_{i+1}. Once all vertices of L_i are popped from Q, Q will contain exactly the set of vertices L_{i+1} and one can start iteration $i+1$ of BFS.

This view of BFS motivated Edsger Dijkstra in 1959 to consider a generalization of BFS which happens to solve SSSP in graphs with nonnegative weights.

Let D be a data structure, a "priority queue" that can support:

1. nonempty: return whether there D is nonempty
2. Insert$(v, d[v])$: insert v with key $d[v]$
3. DecreaseKey(v, d'): If $d' < d[v]$, replace $d[v]$ with d' as v's key in D
4. DeleteMin: delete and return an element v of minimum $d[v]$ among all elements in D.

Using such a data structure, one can generalize BFS. The main differences are that:

- In Dijkstra's all vertices are initially inserted into D with their initial $d[v]$ (0 or ∞); in BFS vertices get inserted into the queue Q only when their distances are first computed,
- Popping the first element from Q is replaced by $D.DeleteMin$,
- no visited markers are needed,
- at relaxation, $D.DecreaseKey$ is called.

Here is the pseudocode:

Algorithm 1: Dijkstra$(G = (V, E, w), s)$

$\forall t \in V, d[t] \leftarrow \infty, \pi[t] \leftarrow NIL$ // set initial distance estimates;
$d[s] \leftarrow 0, \pi[s] \leftarrow s$;
$\forall t \in V, D.insert(t, d[t])$ // D is set of nodes that are yet to achieve final distance estimates;
while $D.nonempty$ **do**
  $x \leftarrow D.DeleteMin$;
  for *all* $(x, y) \in E$ **do**
    if $d[y] > d[x] + w(x, y)$ **then**
      $d[y] \leftarrow d[x] + w(x, y)$ // "relax" the estimate of y;
      $D.DecreaseKey(y, d[y])$;
      $\pi(y) \leftarrow x$;
    end
  end
end

The running time of Dijkstra's algorithm is easy to see, given what we know about BFS. Let $T(n)$ be the time that data structure D takes on nonempty, Insert or DeleteMin when n elements are ever inserted in D. Similarly, let $Dec(n)$ be the time that D takes on DecreaseKey.

The runtime of Dijkstra's excluding the DecreaseKey operations is $O(m + n \cdot T(n))$ because each vertex x is extracted only once from D and it is never reinserted. The total number of DecreaseKey operations is $O(m)$ as DecreaseKey is only called on relax which happens at most twice per edge (once when each of its endpoints is deleted from D; it happens at most once for directed graphs). The total runtime of Dijkstra's algorithm is thus

$$O(mDec(n) + nT(n)).$$

Let's consider three types of data structures that can support the operations needed by D:

1. D is composed of two arrays/dictionaries indexed by V, d and vis and an integer $size$, where $size$ is the number of elements in D, for $v \in V$, $d[v]$ is just the current distance estimate, and $vis[v] = 1$ if v is supposed to be in D and $vis[v] = 0$ if v was deleted from D.

   After setting up D in $O(n)$ time, Inserts aren't needed so take $O(1)$ time, nonempty takes $O(1)$ since we have $size$, DecreaseKey(v, d') simply sets $d[v] \leftarrow d'$ so takes $O(1)$ time, and DeleteMin takes $O(n)$ time since d is unsorted and since we need to go through vis to know which elements are still in D; on DeleteMin we decrement $size$.

   With this data structure, Dijkstra's algorithm on n-vertex, m-edge graphs runs in $O(m + n^2) = O(n^2)$ time.

2. D is an AVL tree or a min-Heap. Here Nonempty, Insert and DeleteMin take $O(\log n)$ time, and DecreaseKey can be implemented by deleting and reinserting with the new key, taking $O(\log n)$ time as well.

   With these data structures, Dijkstra's algorithm runs in $O((m + n) \log n)$ time. This improves upon the $O(n^2)$ time from the previous bullet except for very dense graphs.

3. In 1984, Fredman and Tarjan designed the Fibonacci Heaps data structure which is outside the scope of 6.006. It supports nonempty, DecreaseKey and Insert in amortized $O(1)$ time and DeleteMin in $O(\log n)$ time. With these times, Dijkstra's algorithm runs in $O(m + n \log n)$ time. This improves upon both running times in the above bullets.

In Figure 1 we give an example run of Dijkstra's algorithm.

Now that we have discussed the running time, let us discuss the correctness of the algorithm. Let us recall that in Lecture 11 we proved that when there are no negative cycles, there is always a shortest paths

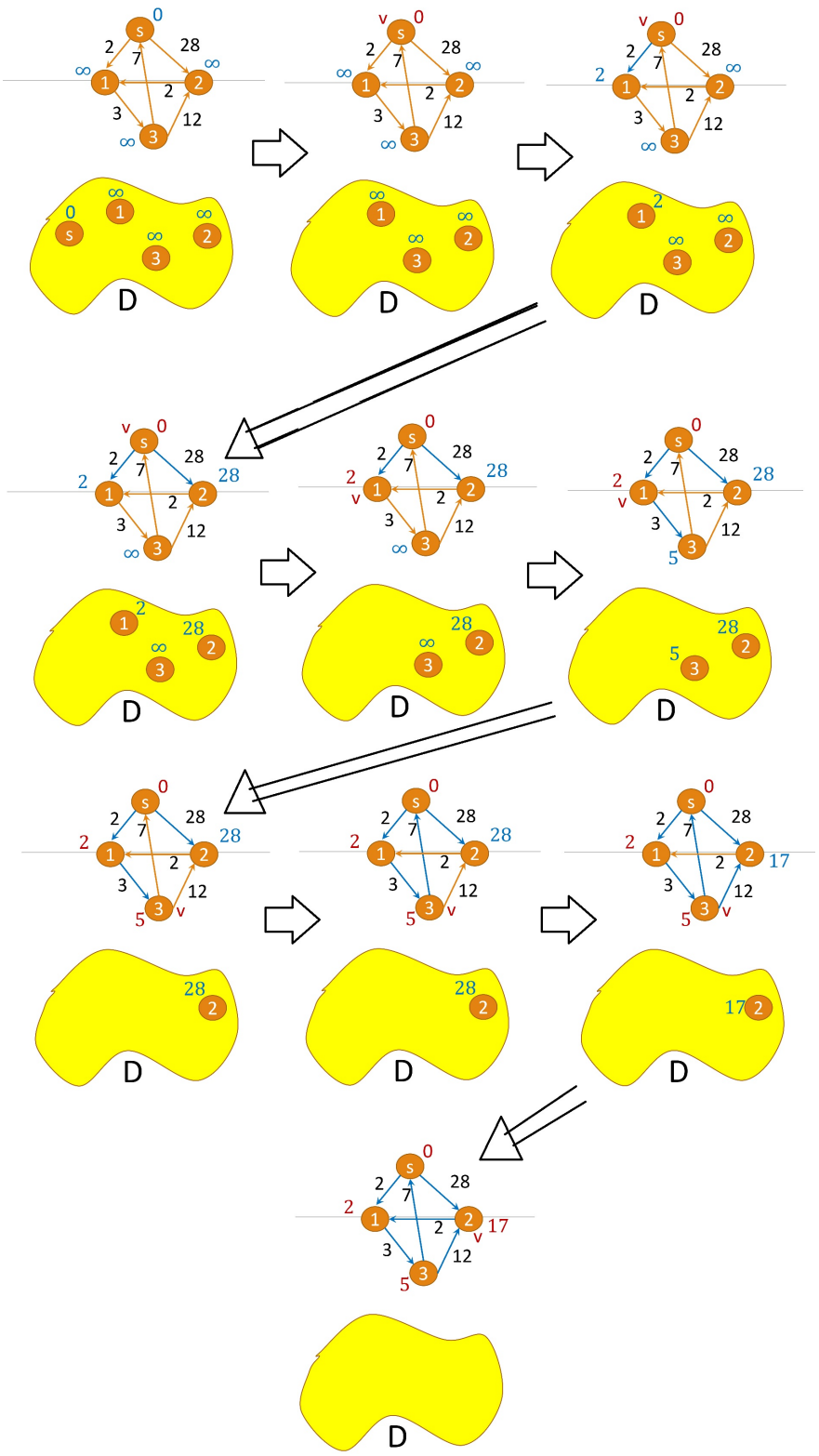

Figure 1: An example run of Dijkstra's algorithm.

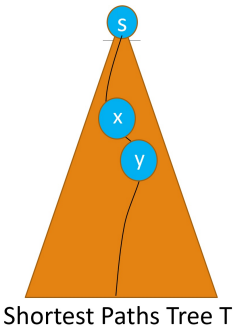

Figure 2: Figure useful for Claim 1.

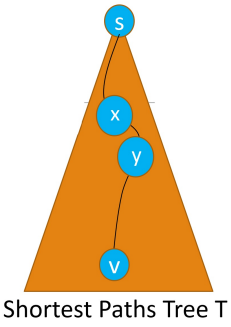

Figure 3: Figure useful for Claim 2.

tree T rooted at the source s so that for every $v \in V$, the path in the tree T from s from v is a shortest path in G from s to v and every edge of T is an edge in G with the same weight.

To prove the correctness, we will prove some claims about the algorithm, while considering a shortest paths tree.

Let T be a shortest paths tree rooted at s.

Claim 1. *Let y be a child of x in T. Suppose that when x is popped from D, $d[x] = d(s,x)$. Then, after popping x and relaxing its edges, $d[y] = d(s,y)$.*

Claim 1 says that if x is deleted from D with the correct distance value, then all its children will get the correct distance estimates after relaxing all edges out of x.

Proof. (See Figure 2.) After x is popped from D, edge (x,y) is relaxed, and after that $d[y] \leq d[x] + w(x,y)$. But since $d[x] = d(s,x)$ and since $d(s,x) + w(x,y) = d(s,y)$ (path in tree is a shortest path), we get

$$d[y] \leq d[x] + w(x,y) = d(s,x) + w(x,y) = d(s,y) \leq d[y],$$

where the last inequality is due to the invariant of edge relaxation that we always overestimate the true distance. This means that $d[y] = d(s,y)$. □

Given Claim 1, we can prove Claim 2 which shows that Dijkstra's algorithm computes all the distances correctly:

Claim 2. *For every $v \in V$, when v is popped from D, $d[v] = d(s,v)$.*

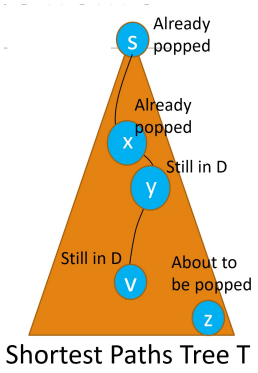

Figure 4: Figure useful for Claim 3.

Proof. Consider some path in T from s down to a leaf. We will prove the claim by induction on the distance from s down this path. Then we will have shown that the claim is true for every node on every path in the tree.

Consider a vertex v on our chosen path, and look at when Dijkstra's algorithm decides to delete v from D. Suppose that at that point (inductively) for every vertex z on the path in T from s to v such that z has already been popped from D, we have that $d[z] = d(s, z)$. (The base case is when $v = s$ and there are no nodes on the path that have been popped.) We will show that $d[v] = d(s, v)$ as well.

Look at Figure 3. Let x be the node closest to v on the s-v path in T that has already been popped from D. Then the node y after x on the path is in D. (Note x and y exist since s has been popped and v is in D.) Since by induction we assumed that $d[x] = d(s, x)$, by Claim 1, $d[y] = d(s, y)$.

Since both v and y are in D but v is to be popped first, $d[v] \leq d[y]$. But then we get

$$d(s, v) \leq d[v] \leq d[y] = d(s, y) \leq d(s, v),$$

where the last inequality is because the weights are nonnegative and y is on the s-v path. Hence we must have that $d[v] = d(s, v)$. □

We have shown that Dijkstra's algorithm correctly computes the distances. We will now show that Dijkstra's algorithm computes the distances in nondecreasing order of their magnitude. In other words, it sorts the vertices by their distance from s!

Claim 3. *If $d(s, v) < d(s, z)$, v will be popped from D before z.*

Proof. For contradiction, let's assume that z is popped from D before v. Look at Figure 4. Let x be the closest vertex to v on the s-v path in T that has already been popped and let y be the node after it; y is in D. (Note x and y exist since s has been popped and v is in D.)

Consider the moment right as z is to be popped. By Claim 2, $d[z] = d(s, z)$ and $d[x] = d(s, x)$ and by Claim 1, $d[y] = d(s, y)$. We have that since the weights are nonnegative, $d(s, y) \leq d(s, v)$. Combining:

$$d[y] = d(s, y) \leq d(s, v) < d(s, z) = d[z].$$

But then we have that y and z are both in D and $d[y] < d[z]$, which is a contradiction since z was supposed to be popped and hence has to have the smallest $d[]$ in D. □

From this claim we see that Dijkstra's algorithm can be used to sort n elements: say we want to sort $x_1, \ldots, x_n$. Then create a graph which is a source vertex s connected to n vertices $\{1, \ldots, n\}$, where $w(s, j) = x_j$. The order in which Dijkstra's pops them from D will be their sorted order. Thus, the $n \log n$ part of

Dijkstra's running time is unavoidable unless one does more with the weights than compare. The m part of the runtime is also unavoidable since one needs to at least read the input.

Finally, it is a good exercise to see why Dijkstra's algorithm does not work on graphs with negative weight edges. This is not surprising, all our proofs needed the nonnegativity of the weights.