1 Dijkstra's Algorithm

Now we will solve the single source shortest paths problem in graphs with nonnengative weights using Dijkstra's algorithm. The key idea, that Dijkstra will maintain as an invariant, is that $\forall t \in V$, the algorithm computes an estimate $d[t]$ of the distance of t from the source such that:

1. At any point in time, $d[t] \geq d(s,t)$, and
2. when t is finished, $d[t] = d(s,t)$.

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

```
∀t ∈ V, d[t] ← ∞ // set initial distance estimates
d[s] ← 0
F ← {v | ∀v ∈ V} // F is set of nodes that are yet to achieve final distance estimates
D ← ∅ // D will be set of nodes that have achieved final distance estimates
while F ≠ ∅ do
    x ← element in F with minimum distance estimate
    for (x, y) ∈ E do
        d[y] ← min{d[y], d[x] + w(x, y)} // "relax" the estimate of y
        // to maintain paths: if d[y] changes, then π(y) ← x
    F ← F \ {x}
    D ← D ∪ {x}
```

We will prove that Dijkstra correctly computes the distances from s to all $t \in V$.

Claim 1. *For every u, at any point of time $d[u] \geq d(s,u)$.*

A formal proof of this claim proceeds by induction. In particular, one shows that at any point in time, if $d[u] < \infty$, then $d[u]$ is the weight of some path from s to t. Thus at any point $d[u]$ is at least the weight of the *shortest* path, and hence $d[u] \geq d(s,u)$.

As a base case, we know that $d[s] = 0 = d(s,s)$ and all other distance estimates are $+\infty$, so we know that the claim holds initially. Now, when $d[u]$ is changed to $d[x] + w(x,u)$ then (by the induction hypothesis) there is a path from s to x of weight $d[x]$ and an edge (x,u) of weight $w(x,u)$. This means there is a path from s to u of weight $d[u] = d[x] + w(x,u)$. This implies that $d[u]$ is at least the weight of the shortest path $= d(s,u)$, and the induction argument is complete.

Claim 2. *When node x is placed in $D, d[x] = d(s,x)$.*

Notice that proving the above claim is sufficient to prove the correctness of the algorithm since $d[x]$ is never changed again after x is added to D: the only way it could be changed is if for some node $y \in F$, $d[y]+w(y,x) < d[x]$ but this can't happen since $d[x] \leq d[y]$ and $w(y,x) \geq 0$ (all edge weights are nonnegative). The assertion $d[x] \leq d[y]$ for all $y \in F$ stays true at all points after x is inserted into D: assume for contradiction that at some point for some $y \in F$ we get $d[y] < d[x]$ and let y be the first such y. Before $d[y]$ was updated $d[y'] \geq d[x]$ for all $y' \in F$. But then when $d[y]$ was changed, it was due to some neighbor y' of y in F, but $d[y'] \geq d[x]$ and all weights are nonnegative, so we get a contradiction

We prove this claim by induction on the order of placement of nodes into D. For the base case, s is placed into D where $d[s] = d(s,s) = 0$, so initially, the claim holds.

For the inductive step, we assume that for all nodes y currently in D, $d[y] = d(s, y)$. Let x be the node that currently has the minimum distance estimate in F (this is the node about to be moved from F to D). We will show that $d[x] = d(s, x)$ and this will complete the induction.

Let p be a shortest path from s to x. Suppose z is the node on p closest to x for which $d[z] = d(s, z)$. We know z exists since there is at least one such node, namely s, where $d[s] = d(s, s)$. By the choice of z, for every node y on p between z (not inclusive) to x (inclusive), $d[y] > d(s, y)$. Consider the following options for z.

1. If $z = x$, then $d[x] = d(s, x)$ and we are done.

2. Suppose $z \neq x$. Then there is a node z' after z on p. (Here it is possible that $z' = x$.) We know that $d[z] = d(s, z) \leq d(s, x) \leq d[x]$. The first $\leq$ inequality holds because subpaths of shortest paths are shortest paths as well, so that the prefix of p from s to z has weight $d(s, z)$. In addition, the weights on edges are non-negative, so that the portion of p from z to x has a nonnegative weight, and so $d(s, z) \leq d(s, x)$. The subsequent $\leq$ holds by Claim 1. We know that if $d[z] = d[x]$ all of the previous inequalities are equalities and $d[x] = d(s, x)$ and the claim holds.

   Finally, towards a contradiction, suppose $d[z] < d[x]$. By the choice of $x \in F$ we know $d[x]$ is the minimum distance estimate that was in F. Thus, since $d[z] < d[x]$, we know $z \notin F$ and must be in D, the finished set. This means the edges out of z, and in particular (z, z'), were already relaxed by our algorithm. But this means that $d[z'] \leq d(s, z) + w(z, z') = d(s, z')$, because z is on the shortest path from s to z', and the distance estimate of z' must be correct. However, this contradicts z being the closest node on p to x meeting the criteria $d[z] = d(s, z)$. Thus, our initial assumption that $d[z] < d[x]$ must be false and $d[x]$ must equal $d(s, x)$.

1.1 Implementation of Dijkstra's Algorithm

Consider implementing Dijkstra's algorithm with a priority queue to store the set F, where the distance estimates are the keys. The initialization step takes $O(n)$ operations to set n distance estimate values to infinity and 0. In each iteration of the while loop, we make a call to find the node x in F with the minimum distance estimate (via, say, `FindMin` operation). Then, we relax each edge leaving x (via `DecreaseKey`). We remove node x (via `DeleteMin`) and add it to D. In total, there are n calls to `FindMin` and n calls to `DeleteMin` since nodes are never re-inserted into F. Similarly, there will be m calls to `DecreaseKey` to relax the edges since each edge will be relaxed at most once.

Depending on how quickly our priority queue can support `FindMin`, `DeleteMin`, and `DecreaseKey` operations, the total runtime of Dijkstra's algorithm is on the order of

$$n \cdot (T_{\texttt{FindMin}}(n) + T_{\texttt{DeleteMin}}(n)) + m \cdot T_{\texttt{DecreaseKey}}(n).$$

We consider the following implementations of the priority queue for storing F:

- Store F as an array:
  Each slot corresponds to a node and stores the distance $d[j]$ if $j \in F$, or NIL otherwise. `DecreaseKey` runs in $O(1)$ as nodes are indexed. `FindMin` and `DeleteMin` run in $O(n)$ as the array is not sorted and we have to go through the whole array. The total runtime is $O(m + n^2) = O(n^2)$.

- Store F as a red-black tree:
  All operations run in $O(\log n)$ time. We implement `DecreaseKey` by deleting and re-inserting with the new key. The total runtime is $O((m+n)\log n)$. If graph G is sparse with few edges, then the red-black tree implementation is faster than the array implementation. However, it can be slower when G is dense with $m = \Theta(n^2)$.

- Store F as a Fibonacci heap:
  Fibonacci heaps are a complex data structure which is able to support the operations `Insert` in

$O(1)$, `FindMin` in $O(1)$, `DecreaseKey` in $O(1)$ and `DeleteMin` in $O(\log n)$ "amortized" time, over a sequence of calls to these operations. The meaning of amortized time in this case is as follows: starting from an empty Fibonacci heap, any sequence of operations that includes a `Insert`'s, b `FindMin`'s, c `DecreaseKey`'s and d `DeleteMin`'s take $O(a+b+c+d\log n)$ time. The total runtime is $O(m+n\log n)$.

To conclude, Dijkstra's algorithm can be very fast when implemented the right way! However, it has a few drawbacks:

- It doesn't work with negative edge weights: we used the fact that the weights were non-negative a few times in the correctness proof above.
- It's not very amenable to frequent updates. Suppose that you had already run Dijkstra's algorithm from a particular point, but one weight in the graph changed. How would you recover from this? Next time, we'll see the Bellman-Ford algorithm, which can be better on both of these fronts.