1 Shortest Path with Edge Lengths (Dijkstra's Algorithm)

1.1 An imprecise implementation

Input: A graph in adjacency list representation and a vertex s. For each edge e, a positive length ℓ_e
Result: For each vertex v a label $d(v)$ which is the distance of v from s
For all vertices u, set $discovered(u) \leftarrow$ false;
Set $d(s) \leftarrow 0$;
Set $discovered(s) \leftarrow$ true;
while *there are undiscovered vertices connected to discovered vertices,* **do**
  Let v be the undiscovered vertex that minimizes $d(u) + c_{\{u,v\}}$, where u is a discovered vertex. This is the cheapest way to leave the set of discovered vertices;
  Set $discovered(v) \leftarrow true$;
  Set $d(v) = d(u) + c_{\{u,v\}}$;
end

Algorithm 1: Shortest Paths with Edge Lengths

The proof of correctness follows from the following lemma:

Lemma 1. *Every vertex that is reachable from s is assigned its shortest path to s as $d(v)$.*

Proof During the run of the algorithm, let S be the set of vertices that have been assigned a distance, *i.e* let S be the set of discovered vertices. We prove by induction on $|S|$ that every vertex of S is assigned the correct distance. Note that eventually S is all vertices reachable from s.

Base case: $S = \{s\}$. In this case obviously the shortest path to s is zero.

IH: Suppose for some $k \geq 1$, when $|S| = k$ we have found the shortest path to all vertices in S.

IS: Suppose v is the $k+1$ vertex to be added to S i.e., $v \notin S$ is about to be added to S with $d(v) = d(u) + c_{\{u,v\}}$. Then by IH u is at distance $d(u)$ from s, so v is at distance at most $d(v)$ from s. For the sake of contradiction, suppose $d(u) + c_{\{u,v\}}$ is not the shortest path to v. Suppose there is a path $P' = (s = u_0, \ldots, u_k = v)$ of cost less than $d(v)$ to v. Then there is some i such that $u_i \in S$, but $u_{i+1} \notin S$. By IH, u_i is assigned the correct distance $d(u_i)$. Therefore, the cost of the sub-path of P' from s to u_i is at least $d(u_i)$. Since costs of all edges are nonnegative, the cost of P' is at least the cost of the subpath from s to u_{i+1}, i.e., at least $d(u_i) + c_{\{u_i,u_{i+1}\}}$. Thus, we must have

$$d(u_i) + c_{\{u_i,u_{i+1}\}} < d(v).$$

But then the algorithm must have considered adding u_{i+1} to S, or in other words, getting to v from s (through u) was not the cheapest way to leave S. That is a contradiction. ■

In order to give a fast implementation of Dijkstra's algorithm, we need a data structure called a heap. This is a data structure that maintains a balanced binary tree of the vertices, where each vertex is given a key value $d(v)$. The data structure maintains the invariant that if v is the parent of u, w in the data structure, then $d(v) \leq \min\{d(u), d(v)\}$.

Input: A graph in adjacency list representation and a vertex s. For each edge e, a positive length l_e
Result: For each vertex v a label $d(v)$ which is the distance of v from s
For all vertices u, set $d(u) = \infty$;
Set $d(s) = 0$;
Initialize a heap H to store all vertices using $d(v)$ as key;
while *H is not empty,* **do**
    Delete the vertex u with the minimum key value $d(u)$ from H;
    for *all edges* $\{u, v\}$ **do**
        if $d(v) > d(u) + c_{\{u,v\}}$ **then**
            $d(v) = d(u) + c_{\{u,v\}}$;
            decrease the key of v in H;
        end
    end
end

Algorithm 2: Shortest Paths with Edge Lengths

To finish our description of the algorithm, we need to explain how to handle the heap operations. We can maintain the initial heap by putting the vertices v in an array where s is the first element and all other elements follow it (in arbitrary order). This encodes the binary tree where each element at location j in the array is the parent of the elements at location $2j$ and $2j + 1$. To delete the minimum element from the heap, we can take the element that is at the end of the array and write it over the element at location 1. We can then compare this element to its children and rearrange the elements as needed to maintain the heap invariant. This requires at most $O(\log n)$ swaps to fix the heap invariant throughout the tree. Similarly, to decrease $d(v)$ for any vertex $d(v)$, we can repeatedly swap v with its parent until the heap invariant property is restored. This also requires at most $O(\log n)$ operations. Thus the total running time of the above algorithm is $O((n + m) \log n)$.