Dijkstra's Algorithm

Exercise 2 asks for an algorithm to find a path of maximum bottleneck capacity in a flow graph G with source s, sink t, and positive edge capacities $c : E \to \mathbb{N} - \{0\}$. A hint is provided suggesting that you use a modified version of Dijkstra's algorithm. The purpose of this note is to review Dijkstra's algorithm and its proof of correctness. You may use this as a template on which to model your solution if you wish.

Dijkstra's algorithm solves the *single-source shortest path* problem for directed graphs with nonnegative edge weights. Given a directed graph $G = (V, E)$ with edge weights $d : E \to \mathbb{N}$ and a source $s \in V$, we would like to find a shortest path from s to every other $v \in V$, where *shortest* means the sum of the weights of the edges along the path is minimum among all paths from s to v.

For $X \subseteq V$, call a path an X-*path* if all nodes on the path except possibly the last lie in X. That is, $s_0, \ldots, s_n$ is an X-path if $s_0, \ldots, s_{n-1}$ lie in X. The last node s_n may be in X or not. Dijkstra's algorithm is greedy, building up a set $X \subseteq V$ inductively. It maintains several data items as it executes:

- A set X of nodes, initially empty. These are the nodes v for which we have already found a shortest path from s to v.
- A priority queue Q containing some nodes in $V - X$. These are the candidates for next inclusion in X. The queue is a min-queue, which means that the item with the least priority value is extracted.
- For each $v \in Q \cup X$, an X-path $p(v)$ from s to v. The priority of $v \in Q$ is the weight of $p(v)$, which we denote by $D(v)$. If $v \neq s$ and $P(v)$ is the immediate predecessor of v on $p(v)$, then $p(v)$ consists of $p(P(v))$ followed by the edge $(P(v), v)$. Thus v need only remember its immediate predecessor $P(v)$, as $p(v)$ can be reconstructed by following the sequence of back-pointers $P(\cdot)$ from v back to s. Moreover, $D(v) = D(P(v)) + d(P(v), v)$.

The following invariants are maintained by the algorithm:

(i) $Q \cup X = \{v \mid \text{there exists an } X\text{-path from } s \text{ to } v\}$.

(ii) For $v \in Q$, $p(v)$ is a shortest X-path from s to v.

(iii) For $v \in X$, $p(v)$ is a shortest path from s to v.

The algorithm proceeds as follows.

1. Set $X := \varnothing$ and $D(s) := 0$. Insert s in Q with priority $D(s)$.
2. Repeat the following until Q becomes empty. Extract the element v from Q with the minimum $D(v)$ value and add v to X. For each edge $(v, w) \in E$,
   (a) If $w \in X$, do not do anything. Go on to the next edge.
   (b) If $w \in Q$ and $D(v)+d(v,w) < D(w)$, reset $P(w) := v$ and reset $D(w) := D(v)+d(v,w)$. (This will cause the priority of w in the priority queue Q to decrease, perhaps requiring some restructuring of Q; we discuss this below.) Otherwise just go on to the next edge.
   (c) If $w \notin Q \cup X$, set $D(w) := D(v) + d(v, w)$, set $P(w) := v$, and insert w in Q with priority $D(w)$.

To prove correctness, we first show that all the invariants are true after initialization (step 1) and are preserved by the loop (step 2).

After step 1, (i) holds because $Q \cup X = \{s\}$ and we can take $p(s)$ to be the 0-length path consisting of just the node s. Moreover, since $X = \varnothing$, this is the only X-path at that point. Property (ii) holds because all edge weights are nonnegative, and $D(s) = 0$, which is as small as possible. Property (iii) holds vacuously.

Now suppose the invariants hold before one execution of the loop body. Say v is the node extracted from Q and added to X in that iteration. The new nodes with an X-path from s are all those reachable in one step from v and not already in $Q \cup X$, and those are all added to Q in 2(c), so (i) is preserved.

For (ii), if $w \in Q$ prior to the execution of the loop body, then the only possibility for a new shortest X-path to w afterward are through v. Step 2(b) checks for this eventuality and updates $P(w)$ and $D(w)$ accordingly if necessary. If $w \notin Q$ prior to the execution of the loop body, then by (i) the only X-paths to w after the execution of the loop are through v, and step 2(c) sets $P(w)$ and $D(w)$ accordingly.

Finally (iii). Just before the execution of the loop body, any path q starting from s and ending at v must leave X for the first time. Thus q has a prefix q' that is an X-path. Say the last two nodes on q' are $x \in X$ and $y \notin X$. By invariant (i), $y \in Q$. Since v was the node extracted from Q, we must have $D(v) \leq D(y)$. The weight of q' is at least $D(y)$ by (ii), and the weight of q is at least the weight of q', therefore the weight of q is at least $D(v)$, the weight of $p(v)$. As q was arbitrary, $p(v)$ is a shortest path from s to v.

Using a heap-based priority queue, the algorithm can be implemented in $O((m+n)\log n)$ time. Step 1 takes constant time. Each iteration of the loop in 2 requires $O(\log n)$ time to extract the min priority node v from Q, or $O(n \log n)$ time over the entire algorithm, and $O(\log n)$ time for each edge (v, w) to add w to Q in 2(c) or to readjust the queue in 2(b) if the decrease of priority causes a violation of heap order, or $O(m \log n)$ time over the entire algorithm. All other operations are constant time per node or edge.