CS5050 Advanced Algorithms

Fall Semester, 2018

Homework Solution 7

Haitao Wang

1. The idea is similar to Dijkstra's algorithm. The only difference is the way we update the value $v.d$ for each vertex v in the algorithm. Specifically, for each vertex v, we let $v.d$ denote the bottleneck weight of the current bottleneck shortest path from s to v that has been found so far. Initially, $s.d = 0$. During the algorithm, we use a priority queue (or heap) Q to store all vertices of G whose bottleneck shortest paths have not be determined yet, and the keys of the heap are still the values $v.d$ for the vertices v of Q.

   As in Dijkstra's algorithm, we repeatedly "extract-min" a vertex u from Q (i.e., u is the vertex of Q with the smallest value $u.d$). Then, we consider the neighbors of u. For each vertex v in the adjacency list of u, if $v.d > \max\{u.d, w(u,v)\}$, then we update $v.d$ by setting $v.d = \max\{u.d, w(u,v)\}$ because we just found a better path from s to v through u. Namely, we replace the "+" operation in the Dijkstra's algorithm by the "max" operation.

   The pseudocode is given in Algorithm 1, where the operation Extract-Min(Q) is to find the vertex u in Q with the minimum value $u.d$ and remove u from Q. The algorithm will find bottleneck shortest paths from s to all other vertices of G and the path information is maintained in the predecessor $v.pre$ for each vertex v (i.e., a "bottleneck shortest path tree" will be produced). To report a bottleneck shortest path from s to t, we only need to follow the predecessor $v.pre$ of the vertices from t back to s.

   The time complexity is the same as Dijkstra's algorithm because we essentially only replace a constant time procedure in each step of the original Dijkstra's algorithm with another constant time procedure.

2. (a) Not necessarily. $\pi(s,t)$ may not be a shortest path any more. Consider the example in Fig. 1(a). The shortest path $\pi(s,t)$ from s to t is $s \to a \to b \to t$. Suppose we increase the weight of each edge by 5 (see Fig. 1(b)). Then, the shortest path from s to t becomes $s \to c \to t$.

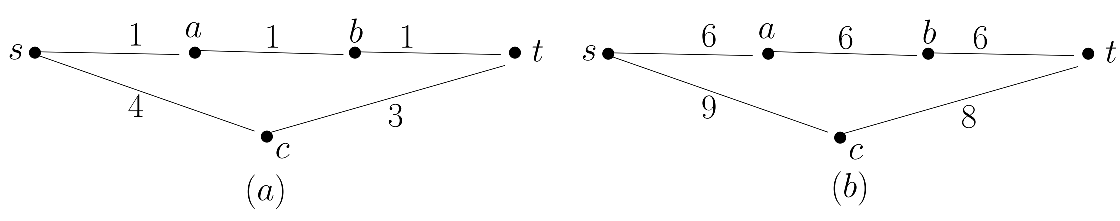

Figure 1: (a) The original graph. (b) The new graph after the edge weights get increased.

Algorithm 1: Bottleneck-Single-Source-Shortest-Paths(G, s)

Input: A graph G, and a source vertex s.

Output: A bottleneck shortest path from s to t. In fact, the algorithm finds bottleneck shortest paths from s to all other vertices and the path information is maintained in the predecessor $v.pre$ for each vertex v.

for *each vertex* $v \in V$ **do**
    $v.d = +\infty$, $v.pre = NULL$;
end
$s.d = 0$;
build a priority queue Q on all vertices of G using the $v.d$ values as the keys;
while $Q \neq \emptyset$ **do**
    $u =$ Extract-Min(Q);
    for *each vertex* $v \in Adj[u]$ **do**
        if $v.d > \max\{u.d, w(u,v)\}$ **then**
            $v.d = \max\{u.d, w(u,v)\}$;
            $v.pre = u$;
        end
    end
end

(b) Yes, T is still a minimum spanning tree. The reason is that any spanning tree of G must have exactly $n-1$ edges. Therefore, as long as all edge weights are changed for the same amount, T is always the minimum one.

Another way to think about this is as follows. Suppose T is the minimum spanning tree produced by running Prim's algorithm on G. Let G' be the graph after the weight of each edge is increased by δ. Now we run Prim's algorithm on G'. Then, the algorithm will behave exactly the same as before on G. Hence, T will be produced again by the algorithm as a minimum spanning tree of G'.

3. We reduce the problem to the problem of computing a minimum spanning tree by introducing weights for the edges of G, as follows.

For each edge of the graph, if it is red, then we set its weight to 2; if it is blue, we set its weight to 1. With the weights of the edges thus defined, a minimum spanning tree of G must be a spanning tree with fewest red edges because every spanning tree of G must have exactly $n-1$ edges.

Therefore, we can compute a minimum spanning tree of G by using Prim's algorithm studied in class. The running time is $O((n+m)\log n)$.