CS5050 Advanced Algorithms

Fall Semester, 2017

Homework Solution 7

Haitao Wang

1. We modify Dijkstra's algorithm in the following way. For each vertex v, we still maintain the current shortest path distance $v.d$ from s, and in addition, we maintain another value $v.count$, which is the number of edges in the current shortest path from s to v. Initially, $s.count = 0$ and $s.d = 0$. During the algorithm, we still use a priority queue (or heap) Q to store all vertices of G whose shortest paths have not be determined yet, and the keys of the heap are still the distance 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 distance value $u.d$). Then, we consider the neighbors of u. For each vertex v in the adjacency list of u, if $v.d > u.d+w(u, v)$, then we update $v.d = u.d+w(u, v)$ and $v.pre = u$, which is the same as before, but here we also need to update $v.count = u.count+1$, because we have just found a shorter path from s to v through u and the number of the edges of the path is $u.count + 1$. If $v.d = u.d + w(u, v)$, in the original Dijkstra's algorithm, we don't need to do anything, because we have found another path whose length is the same as $v.d$. But here we need to compare $v.count$ and $u.count + 1$. If $v.count > u.count + 1$, then we update $v.count = u.count + 1$ and $v.pre = u$ because the new path has fewer edges. Otherwise we do nothing.

   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 optimal paths from s to all other vertices of G and the path information is maintained in the predecessor $v.pre$ for each vertex v. To report an optimal 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 only add an additional constant time procedure in each step of the original Dijkstra's algorithm.

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 $s \to c \to t$.

   (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

Algorithm 1: Optimal-Path(G, s, t)

Input: A graph $G = (V, E)$, and two vertices s and t.
Output: An optimal path from s to t. In fact, the algorithm finds optimal 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* $u \in V$ **do**
    $u.d = +\infty$, $u.count = +\infty$, $u.pre = NULL$;
end
$s.d = 0$, $s.count = 0$;
build a heap Q on all vertices of G whose keys are the values $v.d$;
while $Q \neq \emptyset$ **do**
    $u =$ Extract-Min(Q);
    for *each vertex* $v \in Adj[u]$ **do**
        if $v.d > u.d + w(u, v)$ **then**
            $v.d = u.d + w(u, v)$, decrease-key$(Q, v, v.d)$, $v.pre = u$, $v.count = u.count + 1$;
        else
            if $v.d = u.d + w(u, v)$ *and* $v.count > u.count + 1$ **then**
                $v.pre = u$, $v.count = u.count + 1$;
            end
        end
    end
end

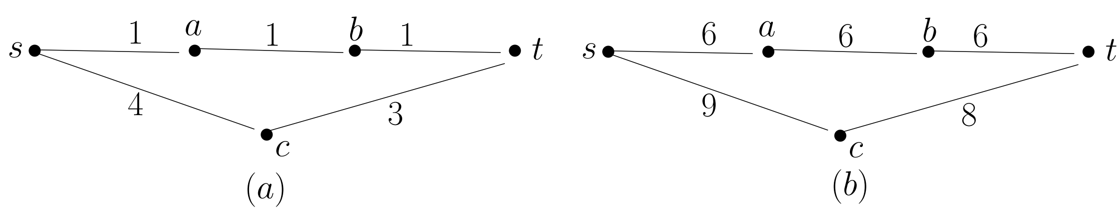

Figure 1: (a) The original graph. (b) The new graph after the edge weights get increased.

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 $O((n + m) \log n)$.