COT 5405 - HW 04

solution

November 17, 2009

1. (a) If all the edge weights are 1, then for each node, edge relaxation will occur only once, so for that node d[u] will be the shortest distance from the source.Moreover, in extract-min function the nodes are extracted in increasing order of d value. Actually in Relax procedure w(u,v) = 1, so d[v] is set to d[u]+1, when v is parent of u.

   Priority Queue of Dijkstra and the queue in BFS in this case is related as follows.-

   (1) At any point in time the BFS queue contains exactly those nodes that the heap in Dijkstra has a d value other than infinity.

   (2) The Dijkstra priority queue at any time has d values that are of the type x, x+1, infinity or x, infinity (for some integer value x)

   (3) The decrease key and delete min operation in Dijkstra can therefore be related one to one with the dequeue and enqueue operations in the BFS queue

   (b) We show that in each iteration of Dijkstras algorithm, $d[u] = \delta(s, u)$ when u is added to S (the rest follows from the upper-bound property). Let $N(s)$ be the set of vertices leaving s, which may have negative weights. We divide the proof to vertices in $N(s)$ and all the rest.

   Since there are no negative loops, the shortest path between s and all $u \in N(s)$, is through the edge connecting them to s, hence $\delta(s, u) = w(s, u)$, and it follows that after the first time the loop in lines $7, 8$ is executed $d[u] = w(s, u) = \delta(s, u)$ for all $u \in N(s)$ and by the upper bound property $d[u] = \delta(s, u)$ when u is added to S. Moreover it follows that $S = N(s)$ after $|N(s)|$

steps of the while loop in lines 4-8.

For the rest of the vertices we argue that the proof of Theorem 24-6 holds since every shortest path from s includes at most one negative edge (and if does it has to be the first one). To see why this is true assume otherwise, which mean that the path contains a loop, contradicting the property that shortest paths do not contain loops.

2. (a)Observe that MAYBE-MST-A removes edges in non-increasing order as long as the graph remains connected. Then, the resulting T is a minimum spanning tree. The correctness of the algorithm can be shown as follows: let S be a MST of G. When an edge e is about to be removed, either $e \in S$ or $e \notin S$. If $e \notin S$,then we can just remove it. If $e \in S$, removing e will disconnect S into two trees. Note that, this does not disconnect the graph. It is clear that no other edge can connect these trees with smaller weight than e, because by assumption S is a MST, and if a larger edge existed that connected the trees the algorithm would have removed it before removing e. Since the graph is still connected, this means a path exists. So, there must be another edge with equal weight that has not been discovered yet, which means we can remove e.

   An adjacency list representation for T would be used for implementation. We can sort the edges in $O(E\ logE)$ like using MERGE-SORT. We can check if $T-e$ is a connected graph by using BFS or DFS in $O(V+E)$. Then, the total running time is $O(E\ logE\ +\ E(V+E))\ =\ O(E^2)$.

   (b) MAYBE-MST-B will not give a minimum spanning tree, that can be proven by a counterexample.

   For the given graph G in figure, the MST would have edges (w, u) and (v, w) with weight 3. MAYBE-MST-B could add edges (u, v) and (v, w) to T , then try to add (w, u) which forms a cycle, then return T (weight 5), since the algorithm takes edges in arbitrary order.

   This idea is similar to Kruskals algorithm, then similary we can use a UNION-FIND data structure to maintain T . For each vertex v we need to MAKE-SET(v). For each edge

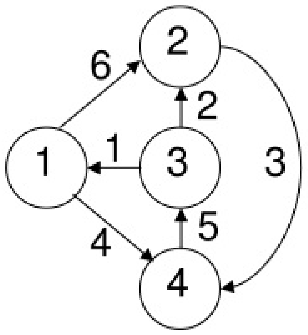
$e = (u, v)$, if FIND-SET (u) $\neq$ FIND-SET (v) then there is no cycle in $T \cup e$, and we UNION-SET(u,v). In total, there are V MAKE-SET operations, $2E$ FIND-SET operations, and $V - 1$ UNION-SET operations. If we use an improved UNION-FIND data structure we can perform FIND-SET in $O(1)$ and UNION-SET in $O(logV)$ time amortized to give a running time of $O(V) + O(E) + O(VlogV) = O(V\ logV + E)$.

(c) MAYBE-MST-C succesfully gives a minimum spanning tree. The algorithm adds edges in arbitrary order to T . If a cycle c is detected removes the maximum-weight edge on c. When an edge e is about to be added, either we form a cycle c or not. Let us assume e is added to some tree T' in T and a cycle is c formed. Then we remove a maximum-weight edge e' from c and $T' - e' + e$ is of less or equal weight than T'. This is because e' is of greater or equal weight than e.In case no cycle is formed, we either just add e to some tree in T or e connects two trees in T . In the second case if e is not the smallest weight edge that connects the trees then we have not discovered it yet, and when we eventually form a cycle we have already shown MAYBE-MST-C performs correctly.
Implementation would use an adjacency list representation for T , similary to part (a). For each edge, we add it to T and check $T \cup e$ for cycles. There can be at most one cycle which we can detect by DFS and return it. Otherwise, when we have no cycles, we are done with this edge. When there is a cycle, we need to put returned cycle, find the maximum-weight edge and delete it from T.
Time complexity can be analyzed as follows: $O(1)$ time for adding an edge. DFS takes $O(V + E) = O(V)$ if we break it as soon as T has a cycle, so number of edges in T at any point is no greater than V . Finding the maximum-weight edge on the cycle would take $O(V)$ time and deleting an edge would take $O(V)$ time. For each edge we added, we will perfom DFS, and might find a cycle, so the total running time becomes $O(EV)$.

3. (a) Lets assume that T is a MST which does not include the minimum weight edge e incident to vertex u. Now, if we add this edge to the MST T, then it will create a cycle. This cycle must

contain an edge e^1 which is also incident to vertex u and $e^1 > e$. Now, if we remove e^1 and keep e, we will get a spanning tree with lower cost. But this is a contradiction. Therefore, any MST must contain e.

(b) If the graph is directed, it is possible for a tree of shortest paths from s and a minimum spanning tree in G not share any edges. A counter example is shown in figure below. MST will have the edges $\{(3,1),(3,2),(2,4)\}$. Shortest path tree rooted at vertex 1 has the edges $\{(1,4),(1,2),(4,3)\}$. However if the graph is undirected, by the cut property minimum cut edge is in MST. According to Dijkstra algorithm, minimum cut edge must be in shortest path tree.



4. Observe that a bitonic sequence can increase, then decrease, then increase, or it can decrease, then increase, then decrease, since a cyclic shift of a bitonic sequence is again bitonic. That is, there can be at most two changes of direction in a bitonic sequence. A bitonic sequence would be a subsequence of any sequence that increases, then decreases, then increases, then decreases. First, we assume our problem has the following property instead of bitonic sequence one: for each vertex $v \in V$, the weights of the edges along any shortest path from s to v are increasing. In that case, in order to find the shortest paths we could simply call INITIALIZE-SINGLE-SOURCE and then just relax all edges one time, going in increasing order of weight. The edges along every shortest path would be relaxed in order of their appearance on the path. Note that, we assume the uniqueness of edge weights to ensure that the ordering is correct. The path-relaxation property in Lemma 24.15 would guarantee that we would have computed correct shortest paths from s to each vertex.

Second, we assume our problem has another property instead of bitonic sequence one: the weights of the edges along any shortest path increase and then decrease. In that case, we could simply relax all edges one time, in increasing order of weight, and then one more time, in decreasing order of weight.Since edge weights are unique, this order would ensure that we had relaxed the edges of every shortest path in order. Hence, the path-relaxation property would guarantee that we would have computed correct shortest paths.

Let us apply the above mentioned ideas to solve our problem, which ensures bitonic sequences. We need to perform four passes to cover all possible bitonic sequences as mentioned earlier. We will relax each edge once in each pass. In particular, the first and third passes relax edges in increasing order of weight, and the second and fourth passes in decreasing order. We ensure the correctness by the path-relaxation property together with the uniqueness of edge weights. So, we have computed correct shortest paths.

The total time for this procedure is $O(V + E\ logV)$ which consists of sorting $|E|$ edges by weight in $O(ElogE) = O(ElogV)$ (since $|E| = O(V^2)$), INITIALIZE-SINGLE-SOURCE in $O(V)$ and $O(E)$ for each relaxation passes. Hence, this will give $O(E\ logV + V + E) = O(V + E\ logV)$ time complexity.

Note: Solutions given are not the only possible solutions, there can be many different solutions.