

CS3500 HW6 Solutions

1 (20 points each)

Run DFS once from each vertex. The graph is singly connected if and only if there are no forward edges and there are no cross edges within a component.

Correctness: “ \Rightarrow ” If the graph G is singly connected, we can prove that there cannot be forward edges. Assume there exists a forward edge (u, v) in G , then there must be a path p down the tree separate from (u, v) which has been found in order for (u, v) to be labelled “forward”. But p and the edge (u, v) are two vertex-disjoint paths from u to v and this contradicts the fact that G is singly connected. Therefore, there are no forward edges in G .

Similarly, assume there exists a cross edge (u, v) within a component, then u and v must share a common ancestor s , the root of the DFS tree, so there are two vertex-disjoint paths from s to u : one through v and the other not. This is a contradiction. So there are no cross edges within a component.

“ \Leftarrow ” Assume there are no forward edges and no cross edges within a component. If G is not singly connected, then there must exist two vertices which have two disjoint paths from one to the other. Choose two such vertices u and v that one of their paths is the shortest among such vertices. Our algorithm will eventually run DFS from u and this will produce a path p from u to v in the DFS forest.

- If p is the shorter path, then there will be another path p' in the same tree from u to v . Let the last vertex before v on path p' be x , then (x, v) is a cross edge in this component.
- If p is not the shorter path, then the shorter path p' will either be a single edge (u, v) , a forward edge, or contain the edge (x, v) , which is again a cross edge.

Both cases contradict our assumption, so the graph G has to be singly connected.

Time: For each vertex, we run DFS to classify each edge, so the running time is $O(E)$. Since we have $|V|$ vertices, the total running time will be $O(VE)$.

2 (20 points)

We can run DFS on G . If a vertex has no parent in the DFS forest, then we color it with RED, otherwise, color it with the opposite color of its parent. If a back edge is encountered, check if its two vertices have the same color. If yes, the graph is not 2-colorable, reject and halt. If the algorithm ends without rejection, the graph is 2-colorable.

Correctness: According to Theorem 22.10, every edge in an undirected graph is either a tree edge or a back edge. In our algorithm, all tree edges are assigned two colors on their vertices. When the algorithm ends without rejection, all back edges are also assigned two colors. So for any edge (u, v) in G , $C(u) \neq C(v)$, i.e., G is 2-colorable. If the algorithm rejects at one point, say, a back edge (u, v) with RED on both u and v is encountered, then there must be a separate path p from v to u in the DFS forest. According to our coloring assignment, p must have odd number of vertices in order for u and v to have the same color. Furthermore, p and edge (u, v) form a cycle. We know that a cycle with odd number of vertices is not 2-colorable. Therefore, G cannot be 2-colorable since it contains such a cycle.

Time: The only modification we need to do to the original DFS is to add BLUE and RED assignment to each vertex according to its parent's color. The running time is still $O(V + E)$.

3 (20 points)

Kruskal's algorithm as given in the book requires $O(E \lg E)$ time:

- $O(V)$ to initialize
- $O(E \lg E)$ to sort the edges by weight
- $O(E\alpha(E, V))$ to process edges

If all edge weights are integers ranging from 1 to $|V|$, we can use counting sort (instead of a more generally applicable sorting algorithm) to sort the edges in $O(V + E) = O(E)$ time. (Note that $V = O(E)$ for a connected graph.) This speeds up the whole algorithm to take only $O(E\alpha(E, V))$ time; the time to process the edges, not the time to sort them, now is the dominant term. Knowledge about the weights won't help speed up any other part of the algorithm since nothing besides the sort uses the weight values.

If the edge weights are integers ranging from 1 to a constant W , we can again use counting sort, which again runs in $O(E)$ time ($O(W + E) = O(E)$)

because W is a constant), so we get the same asymptotic upper bound as above.

4 (20 points)

Let v be the new added vertex and T be the original MST with root s . Add the lightest incident edge to T and we get a new spanning tree T' . For each new edge left, say, (u, v) , add it to the current tree, which must result in a cycle; then remove the edge with the maximum weight on the cycle. Repeat this until all new added edges are processed. To find the maximum-weighted edge in a cycle, first find the path p from v to s and the path p' from u to s . From the root s down, compare the vertices appearing on p and p' . The first common vertex x on both paths is the least common ancestor of u and v . Furthermore, the path $x \rightarrow u$, $x \rightarrow v$ and edge (u, v) form a cycle. The maximum-weighted edge on this cycle is the maximum of edges on path p and p' , and edge (u, v) .

Correctness: We can easily see that the new spanning tree we obtain after processing each new edge is in fact the MST for the current stage. This invariant holds true until all new edges are processed and therefore our algorithm produces the new MST.

Time: For each new edge, we need to find the maximum-weighted edge on the resulting cycle. This requires a search on the MST, which takes $O(V)$ time. If there are k new edges, the total running time of the algorithm will be $O(kV)$.

5 (20 points)

Solution 1: To find the most reliable path between s and t , run Dijkstra's algorithm with edge weights $w(u, v) = -\lg r(u, v)$ to find shortest paths from s in $O(E + V \lg V)$ time. The most reliable path is the shortest path from s to t , and that path's reliability is the product of the reliabilities of its edges.

Correctness: Because the probabilities are independent, the probability that a path will not fail is the product of the probabilities that its edges will not fail. We want to find a path $p : s \rightarrow t$ such that $\prod_{(u,v) \in p} r(u, v)$ is maximized. This is equivalent to maximizing $\lg(\prod_{(u,v) \in p} r(u, v)) = \sum_{(u,v) \in p} \lg r(u, v)$, which is again equivalent to minimizing $\sum_{(u,v) \in p} (-\lg r(u, v))$. (Note: $r(u, v)$ can be 0 and $\lg 0$ is undefined. So in this algorithm, we define $\lg 0 = -\infty$.) Thus if we assign weights $w(u, v) = -\lg r(u, v)$, we have a shortest path

problem.

Since $\lg 1 = 0$, $\lg x < 0$ for $0 < x < 1$, and we have defined $\lg 0 = -\infty$, all the weights w are nonnegative, and we can use Dijkstra's algorithm to find the shortest paths from s in $O(E + V \lg V)$ time.

Solution 2: Modify the original Dijkstra's algorithm to maximize the product of reliabilities along a path instead of minimizing the sum of weights along a path.

In Dijkstra's algorithm, use the reliabilities as edge weights and substitute

- max (and EXTRACT-MAX) for min (and EXTRACT-MIN) in relaxation and the queue
- \times for $+$ in relaxation
- 1 (identity for \times) for 0 (identity for $+$) and $-\infty$ (identity for min) for ∞ (identity for max)

Here is the new RELAX procedure:

RELAX-RELIABILITY(u, v, r)

1. **if** $d[v] < d[u] \cdot r(u, v)$
2. **then** $d[v] \leftarrow d[u] \cdot r(u, v)$
3. $\pi[u] \leftarrow u$

This algorithm is isomorphic to the one above: it performs the same operations except that it is working with the original probabilities instead of the transformed ones.

6 (20 points)

Since the shortest distance from the source s to any vertex is always equal or less than $W(V - 1)$ (the longest path has at most $V - 1$ edges and each edge has a maximum weight of W), we can set up an array A of $W(V - 1)$ pointers with each $A[i]$ pointing to a list of vertices whose distance from s is i . We also need a separate list for vertices whose distance is ∞ . Then we can just process each vertex in each $A[i]$. For DECREASE-KEY, we just need to delete the vertex from its current location in A and add it to the corresponding new location $A[k]$, which both take $O(1)$ time. Since the smallest element among current elements is always greater than the smallest

element in the previous iteration and the maximum element is no more than W , the amortized cost of EXTRACT-MIN is $O(W)$. Therefore, the total runtime of our algorithm is $O(V) \cdot \text{EXTRACT-MIN} + O(E) \cdot \text{DECREASE-KEY}$, i.e., $O(VW + E)$.