Neha Maddali

Problem 1:

We know an undirected graph is acyclic which is a forest iff a DFS yields no back edges. These back edges are the edges(u,v) that connect vertex u to an ancestor v in a depth-first tree. So if there are no back edges that means there are only tree edges which means there is no cycle. Thus, just run DFS. If there is a back edge found, then there is a cycle. So the complexity is O(V) and not O(V+E). In an acyclic undirected forest, $|E| \leq |V|+1$. So if there is a back edge, it must be found before seeing |V| distinct edges.

Problem 2:

Let G=(V,E) be a graph where G has a node corresponding to every variable xi and there is an edge (xi, xj) if xi = xj is an equality constraint. We can do a DFS of G to find all its connected components. For each node xi, if it is in the kth connected component of G, we assign it a mark k. Now, for each inequality constraint $xi \neq xj$ check to see if xi and xj occur in different connected components of G. If there is some inequality constraint $xi \neq xj$ such that xi and xj occur in the same connected component of G, then we report that the constraints are unsatisfiable. If there is no such constraint, then we report satisfiable.

The equality constraints form an equivalence class. If all the constraints are satisfiable, then, there can be no inequality constraint $xi \neq xj$ such that xi and xj are in the same equivalence class; this is because such a constraint is not satisfiable, which leads to a contradiction. Therefore, if all the constraints are satisfiable, then the algorithm behaves correctly.

Suppose that all the constraints are not satisfiable; we claim that this can only happen if there is an inequality constaint $xi \neq xj$ where xi and xj lie in the same equivalence class. Suppose this is not the case, and that for all inequality constraints $xi \neq xj$, xi and xj lie in different connected components. Then, an assignment that assigns a value k to all the variables in connected component k satisfies all the constraints, thus leading to a contradiction. Therefore, if all the constraints are not satisfiable, then, the algorithm behaves correctly as well

Problem 3:

a) Implement an algorithm that linearizes DAG and finds costs in reverse topological order.

```
cheapestCost(G)
        DFS(G,u) //where u is a randomly picked vertex
        (v1, v2…vn)
        for i = n to 1:
                cost[vi] = price(vi)
                for all (vi, vj) ∈ E
                        if cost[vj] < cost[vi]
                                cost[vi] = cost[vj]
```

For this algorithm, consider that the vertices of DAG are in topological order. $v1,...v_n$ will be the linearized order. Vertices reachable from any vertex will be among the vertices which are “after” v_k in the linearized order. After updating the costs for vertices v_{k+1}, … v_n, the cost of v_k will be the minimum *cost* of vertices connected to v_k including itself. Any path from vertex v_k will be through its adjacent vertices. The time for linearizing a DAG is

linear. So to find minimum cost, we visit each edge at most once, therefore, the time is O(V+E) for cheapestCost(G).

b) cheapestCost(G)

```
DFS(G^R, u) //let u be a randomly picked vertex
post[C] = max(post(u)) //for all u in C
for all c in C //where C is the set of all strongly connected components
        price[c] = min(price(u))
(c1, c2, …, cn)
for i = n to 1
        Cost[ci] = price[ci]
        for all (ci, cj) ∈ E
                if cost[cj] < cost[ci]
                        cost[ci] = cost[cj]
```

In a directed graph, the cost of any two nodes in the same strongly connected component will be the same since both are reachable from each other. So it will be enough to run the previous algorithm on the DAG of the strongly connected components of the graph. For a node corresponding to, for example, strongly connected component C, we take $price_C = min(price_u)$ for all u in C. Costs of all u in C will be $price_C$. When the strongly connected components are found, meta-nodes can be topologically sorted by arranging them in decreasing order of their highest post numbers.

Problem 4:

a) In this case, the given minimum spanning tree T is still an MST. To see this, we can consider a run of Kruskal's algorithm which produced T. All the same decisions would be made when $\hat{w}(e) > w(e)$, and so the same tree will be produced.

b) Let e = (u,v). Add e to T which will create a unique cycle which we can find doing BFS in T U {e} starting from u and ignoring weights. This takes O(|T|) = O(|V|) time. Now let's remove the maximum weight edge of that cycle O(|V|).

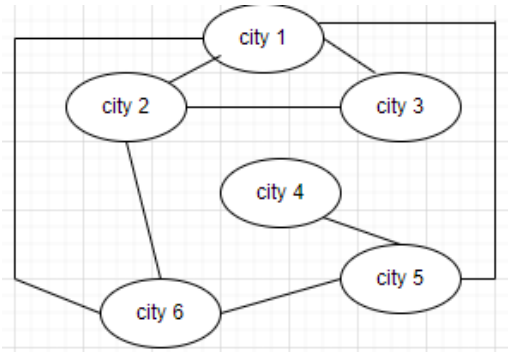
c) In this case, the given minimum spanning tree T is still an MST. Let T be the MST and W be the weight of T. Let e = (u,v), and let the decrease in weight be $\hat{w}(e)$. So the new cost of T becomes $W-\hat{w}(e)$. Let T_u and T_v be the subtrees obtained by removing e. Now if T does not remain an MST, then clearly any new MST T' must contain e, because if it didn't, then since $w(T') < W - \hat{w}(e) < W = w(T)$, this contradicts the fact that T was an MST with the original weights. But if T' contains e, then the only way it can have a cost lower than $W - \hat{w}(e)$ is by either connecting the nodes of T_u with less weight than $w(T_u)$ or by connecting the nodes of T_v with less weight than $w(T_v)$. But T_u and T_v are both minimum spanning trees for their node sets, otherwise, T would not have been minimal to begin with. So T' cannot have less weight than $W - \hat{w}(e)$

d) Let e = (u,v) and let T_u and T_v be the subtrees obtained by removing e. By doing BFS, ignoring weight edges, from u and from v, we can determine which vertices are in T_u and which are in T_v in the time O(V+E). Assume we have marked each node with its membership. Now examine each edge, and keep the minimum weight edge e' with one endpoint in T_u and the other in T_v. This can be done in O(|E|) time. So the total time is O(V+E).

Problem 5:

We can think of this as a graph traversal problem. Each airport acts as a node to the graph. Each flight forms a connecting edge in the graph. The airtime of flight is the respective weights of that edge. The only limitation we have is that the difference between departing time of the next flight and arrival time of the previous flight should be greater than or equal to c(a) which is the minimum connecting time. The traversal will come to an end when paths from source to the destination airport is determined and the weights of such paths are allocated. If a node is visited once, then the cycle formed must be detected and eliminated. So, this problem can be solved using DFS. The time complexity for DFS in this case is O(n+m) where n is the number of nodes (or airports) and m is the number of edges (or flights). If there is no direct connection or flight from one node to another, then the shortest path algorithm or Dijkstra's algorithm should be considered.

Here's an example



If a person needs to go from City 6 to City 3, because there are no direct flights, therefore, shortest path algorithm or Dijkstra's algorithm will come into existence.

Problem 6:

a) Every other spanning tree has a maximum edge cost at least as large. So, an MST is also a minimum bottleneck spanning tree. But the reverse is not true. Consider the graph: G = C -- 4 ← A -- 4 → B -- -1 → C; MST(G) = {(A,B),(B,C)}. Consider the spanning tree {(A,B), (A,C)}. It doesn't increase the bottleneck so it is a minimum bottleneck spanning tree, but isn't an MST.

b) We can map this to a single-source shortest path problem by creating a new graph G' with the same vertices and edges as G but with a negative weight function. Now run the single source shortest paths algorithm for DAG to find the shortest paths in O(V+E). This algorithm relaxes the edges of G' in topologically sorted order only once. Creating G' is a simple process and only requires O(V+E) time to iterate over all the edges and vertices to create the new graph and weight function. Topologically sorting the edges takes O(V+E) since topological sort is done using a modification of the DFS algorithm. Finally, relaxing all the edges once only takes O(E) time. So overall, the runtime is O(V+E).

c) First find the bottlenecks for all the paths until that point, then find the minimum of all such bottlenecks.
dist(v) = min(dist(v), max(dist(u), weight(u,v))) $\forall u \in X, v \in V - X$

Start by inserting (∞, u) u ∈ V in a min-heap. Decrease the key of the source to -∞. Then in each iteration, extract the minimum from the heap, add it to set X and update the keys for the adjacent vertices according to the distance formula. So the main thing we changed from Dijkstra's algorithm is the scoring formula which now involves some constant time operations for min and max. Therefore, the running time of the algorithm described is O(ElogV)