Homework Assignment #6

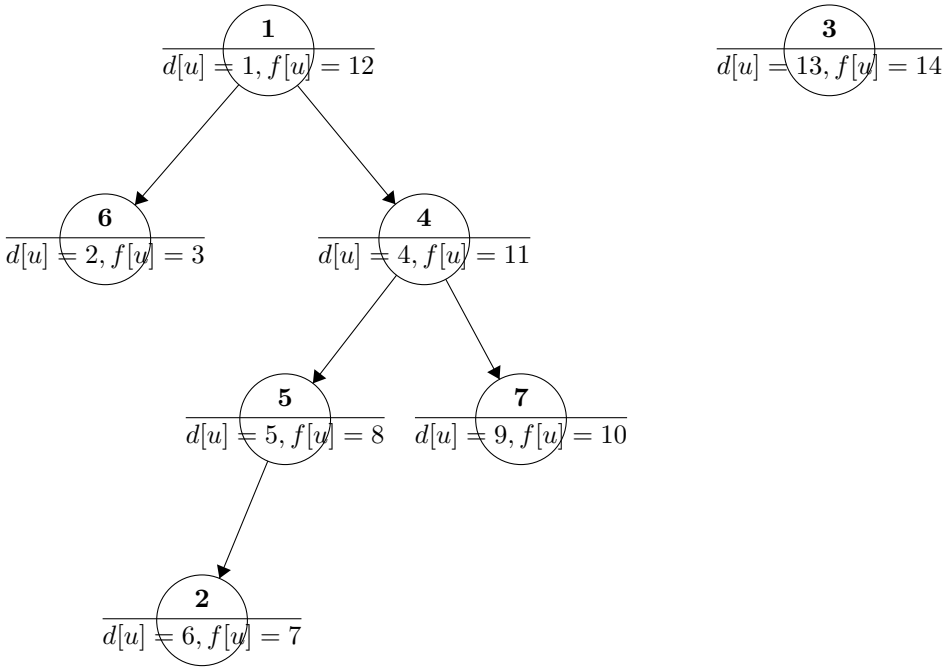
Quan Xu, Zijin Zhang

Due March 28, 2019, by 5:30 p.m.

Question 1.

Quan Xu wrote the solution and Zijin Zhang read this solution to verify its clarity and correctness.

(a)



(b)

number of Back Edges: 0
number of Forward Edges: 2
number of Cross Edges: 5

(c)

By part (b), we know that there is no back edge found by the DFS in part (b), so graph G is acyclic by lemma 22.11 from CLRS P614. Since G is acyclic, we can apply *Topological Sort* on graph G:

1) call $DFS(G)$ to compute finishing time $v.f$ for each vertex v
2) as each vertex is finished(DFS_color is black), insert it onto the front of a linked list
3) return the linked list of vertices

After applying *Topological Sort* on G, we can get a topologically sorted vertices appear in reverse order of their finish time, by theorem 22.12 in CLRS P614. So in the topologically sorted vertices linked list L_t, let

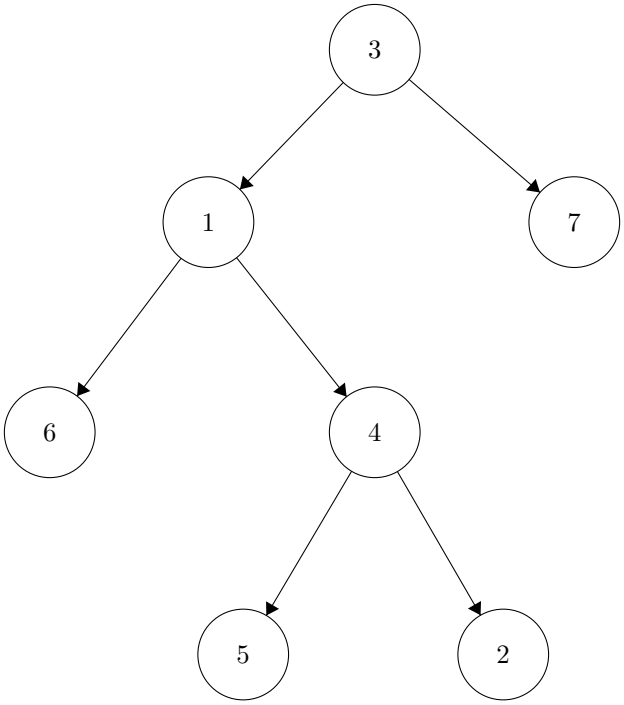
V_i be the vertex in L_t where $i \in 1, 2, 3, ..., n$ is the index. Assume there are n vertices in total and the index starts at 1. For i and j such that $1 \leq i < j \leq n$, we have $f[v_i] > f[v_j]$ which means that v_j is finished before v_i. So v_i could be an ancestor of v_j if there is an edge e_{ij} from v_i to v_j in original graph G. Thus, the course that is represented by v_i is a prerequisite for the course is represented by v_j if such edge e_{ij} exists in G. So the prerequisite course is located before the corresponding courses in L_t. So we could take the course in L_t without violating the prerequisite requirement of each course. Thus in L_t, all the prerequisite courses are located before the corresponding courses, which means it is possible to take all the courses in a sequential order that statisfirs all the prerequisite requirement.

(d)

After applying *Topological Sort* on G and combining the DFS of Part (a), we can get a topologically sorted list that vertices appear in reverse order of their finish time, by theorem 22.12 in CLRS P614:

Course 3 → Course 1 → Course 4 → Course 7 → Course 5 → Course 2 → Course 6

(e)



Question 2.

Zijin Zhang, Quan Xu wrote the solution and Quan Xu, Zijin Zhang read this solution to verify its clarity and correctness.

We will use undirected unweighted graph and use *Breadth First Search* as graph algorithm to implement it.
Assume we have m constraint over n variables. We treat n variables as n vertices.
We also augment $BFS(G, S)$

1) We set a new attribute, $v.root$, for each vertex in G.
2) We set $S.root$ as itself since the root of $BFS\ Tree$ is the starting vertex S.
3) For each vertex v that is discovered during the BFS process, we set $v.root$ as the starting vertex S.
4) The rest procedures of BFS remains unchanged.

Setting $v.root$ as the starting vertex for each vertex v takes constant time and this could be complete during each discover. So the total running time of BFS is unchanged and remains as $\mathcal{O}(m+n)$.

Algorithm:

1) Loop over all n variables, and then creating a graph, G, which contains n vertices and 0 edge.

Loop over all m constraints and do the following operation for **all equality constraint**:

For each *equality constraint* $x_i = x_j, 1 \leq i \neq j \leq n$, we create an unweighted and undirected edge e_{ij} that connect x_i and x_j.

2) After step (1):

We have a new version of graph, G, that contains some edges about all *equality constraints*. Now we do augmented $BFS(G, S)$ on graph G, where S is a randomly chosen starting vertex. Since BFS can be used to find the connected components of an undirected graph, so we can get all connected components after doing BFS on G. Also, all vertices on each connected component have same *root value*(vertices on different connected components have different *root value*).

3) Loop over all m constraints and do the following operation for **all inequality constraint**:

For each *inequality constraint*, $x_i \neq x_j, 1 \leq i \neq j \leq n$, we compare $x_i.root$ with $x_j.root$. If $x_i.root \neq x_j.root$, we continue to check the next *inequality constraint*. If $x_i.root = x_j.root$, this indicates that x_i and x_j are in same connected component, which means $x_i = x_j$ by step 2), which violates the constraint; so we return(output) NIL and the algorithm terminates.

4) If the algorithm does not return NIL and does not terminates in step (4):

We assign an integer to each variable such that this assignment does not violate any of the constraints in this list. And for each connected components, the integer values we assigned to each vertex must be different across different connected components(different connected component has different integer value and all vertices in one connected component have same *integer value*). Then we output integer assignments for all n variables, $\{x_1, x_2, ..., x_n\}$.

Now consider the worst-case time complexity of this algorithm.

In worst-case scenario, assume that all constraints are satisfied and the algorithm will output the assignment of integers to all the variables.

1) Creating graph G takes at most $\mathcal{O}(n)$ where n is number of vertices.

For each iteration, creating a new edge e_{ij} and connecting x_i and x_j takes constant time. There are at most m iterations since there are at most m constraints, so in $\mathcal{O}(m)$.

So step (1) take $\mathcal{O}(n+m)$ in total.

2) We do augmented BFS on graph G. Since there are at most n vertices(since n variables) and m edges(since m constraints) in graph G, then it takes at most $\mathcal{O}(m+n)$ in total.

3) For each iteration, comparing $x_i.root$ with $x_j.root$ takes constant time. In worst-case situation, this step will not return and terminate early. So there are at most m iterations. So step (2) takes at most $\mathcal{O}(m)$ in total.

4) For step (4), each value assignment and output operation of each variable will take constant time $\mathcal{O}(1)$ respectively, and there are at most n vertices. So the total running time for step (4) is at most $n \times \mathcal{O}(1) \in \mathcal{O}(n)$.

Therefore, the worst-case running time of this algorithm is $(\mathcal{O}(n+m)+\mathcal{O}(m+n)+\mathcal{O}(m)+\mathcal{O}(n)) \in \mathcal{O}(m+n)$.

Question 3.

Zijin Zhang wrote the solution and Quan Xu read this solution to verify its clarity and correctness.

Proof. By Contradiction.

Assume that there is at least one minimum spanning tree T of graph G that contains e_{max}.

Since T is a minimum spanning tree as we assumed, remove e_{max} from T will cause the graph T being

divided into two components T_1 and T_2. Since it is given that for every edge $e \in E$ there is a cycle of G that contains e, we can definitely find another edge e' that could reconnect the two components T_1 and T_2 and create another minimum spanning tree T'. Since the edge e_{max} we remove from T is the edge with the maximum weight, the weight of newly added edge e' must be less than the weight of e_{max}. So the total weight of the new minimum spanning tree T' must be less than the weight of T, too. This is contradict with our assumption that T is not the minimum spanning tree in this case. Therefore, our assumption is incorrect and it is proved that no minimum spanning tree of G contains e_{max}. □