CSC263 Fall 2017 Assignment 4

Name: Ruijie Sun, Guanchun Zhao
SN: 1003326046, 1002601847

Q1. SOLUTION

(a) We can use a directed graph with n vertices and n weighted edges to model this problem. In this directed graph, each node is a statement with a Boolean attribute which is set as null initially and an arbitrary edge (u,v) with weight 0 indicates that statement u says that statement v is True. And an arbitrary edge (u,v) with weight 1 indicates that statement u says that statement v is False.

(b) If there exists cycles in the graph and total weight of any cycle is odd number, then it will form a paradox. If there doesn't exist cycle or total weight for all cycles are even number, then it won't form paradox.

(c) Define transformation function

$$\delta(u,(u,v))$$

(u,v).weight \ v	0	1
0	0	1
1	1	0

Firstly, we do regular DFS(G), and by detecting all the back edges, we make a set to store all the vertices that there is a cycle starting from. And reset the graph as initial.

Secondly, for every vertex s in the set we got above, we set its Boolean attribute as 0 which indicates that we assume this node(statement) is true. Then we do the DFS-VISIT from vertex s, whenever we visit a node v from node u, without considering this node's color, we can get a Boolean value from transformation function above. If node v's Boolean attribute is null, we set the Boolean value we get from last step to node v's Boolean attribute. If node v's

Boolean attribute is not null, we check if v's Boolean attribute is equal to the Boolean value from transformation function. If they are equal, there is nothing to do. Otherwise, we find a contradiction. So we return "there is a paradox." Every time we finish a DFS-VISIT, we reset the graph as initial.

If we finish the all DFS-visit without any contradiction, return "there is no paradox."

(d) The worst case happen when there is no paradox, the DFS we do in beginning costs O(V+E), then the other DFS-VISIT we did costs O(V+E). Compared to original DFS-VISIT, setting and checking Boolean attribute cost constant time for each node, thus the total run time is still $\in O(V + E) = O(2n) = O(n)$

Q2. SOLUTION

(a) We can use a directed graph with n vertices and m edges to model this problem. In this directed graph, each vertex is a kid and an arbitrary edge (u,v) indicates that kid u hates kid v.

(b) No. There is no valid arrangement when there is a series of kids, i,j...k where kid i hates kid j, kid j hates... and kid k hates kid i. In other words, these vertices in graph form a cycle, in this case, there must happen one throwing.

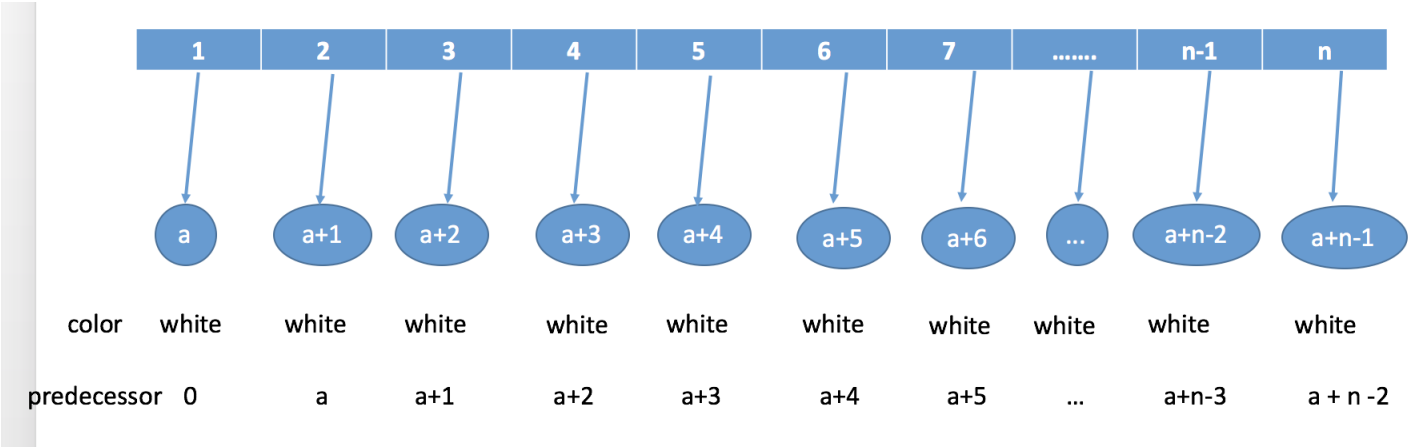
(c) Given that an valid arrangement is possible, we can apply Topological Sort to the graph G we get from part(a) (the graph G is a DAG). In practice, we run DFS(G), when a vertex finished, we output it. And the arrangement order is the vertices output in reverse topologically sorted order. And as we have learnt in class, the worst-case runtime $\in O(n+m)$.

(d) Pseudo code:

```
#G is graph we create in part(a), L is straight line we get from part(c).

def find_number_of_rows(G,L):
    # add an attribute called row to each vertex in L, initialized to 1
    for u in L:
        u.row = 1

    for u in L:
        for v in G.Adj[u]:
            if u.row >= v.row:
                v.row = u.row + 1

    return max([ u.row for u in L])
```

The worst case happens when $m = n-1$, adding and initializing attribute "row" to each vertex cost $O(1)$, so in total they cost $O(n)$. And for the nested loop, since the the number of edges = m = n-1, so it costs $O(n)$. The last part $max([u.rows for u in L])$ costs O(n). Thus the runtime in worst case $\in O(n)$.

Q3. SOLUTION



Initialization: When we initialize the data structure, firstly, we call Make-Set(i) for i = a,...,b. Then we create an Array with n slots where i-th slot stores the pointer to the node with (a + i - 1) as picture above shown. Besides, we add two attributes for each node: color and predecessor. We set color of each node as white and we set 0 as predecessor of node a. For node a + i, we set predecessor as a + i - 1, for i = 1,2,...,n-1. Since each Make-Set() cost O(1), setting attributes cost constant time and creating array and setting pointer cost O(n), initialization costs O(n) in total.

Deletion: When we delete i from S, there are 4 cases.
Case 1: $i < a$ or $i > b$, there is no effect

Case 2: $i = a$, we find the target node through the pointer in first slot of array, if the color of node is black, there is nothing to do, otherwise, we change the color of node from white to black which means this one is deleted from S. Then we check the color of node which is associated with the pointer in second slot, if color is white, there is nothing to do, otherwise, we Union(a, a+1) using Union-by-rank and path-compression. And we update the predecessor attribute of new root by choosing smaller predecessor from root of a and root of (a+1).

Case 3: $i = b$, we find the target node through the pointer in n-th slot of array, if the color of node is black, there is nothing to do, otherwise, we change the color of node from white to black which means this one is deleted from S. Then we check the color of node which is associated with the pointer in (n-1)-th slot, if color is white, there is nothing to do, otherwise, we Union(b, a + n - 2) using Union-by-rank and path-compression. And we update the predecessor attribute of new root by choosing smaller predecessor from root of b and root of (a + n - 2).

Case 4: $a < i < b$, we find the target node through the pointer in (i - a + 1)-th slot of array, if the color of node is black, there is nothing to do, otherwise, we change the color of node

from white to black which means this one is deleted from S. Then we check the color of node which is associated with the pointer in (i - a)-th slot, if color is white, there is nothing to do, otherwise, we Union(i, i - 1) using Union-by-rank and path-compression. And we update the predecessor attribute of new root by choosing smaller predecessor from root of i and root of (i - 1). After that we check the color of node which is associated with the pointer in (i - a + 2)-th slot, if color is white, there is nothing to do, otherwise, we Union(i, i + 1) using Union-by-rank and path-compression. And we update the predecessor attribute of new root by choosing smaller predecessor from root of i and root of (i + 1).

Predecessor: When we need to return the predecessor in S of i, there are three cases.
Case 1: $i \leq a$, return 0.

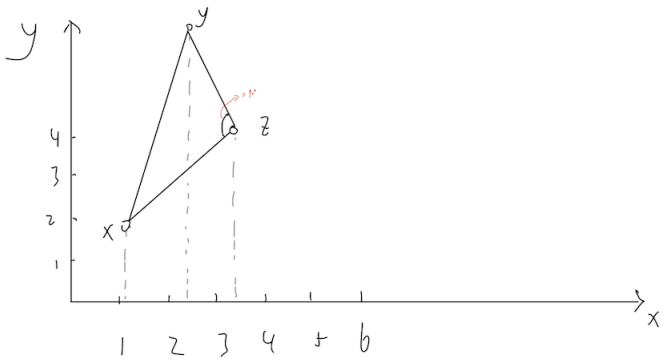
Case 2: $i > b$, we can find the target node through the pointer in n-th slot of array. And if the node' color is white, return b. Otherwise, return Find-Set(b).Predecessor.

Case 3: $a < i \leq b$, we can find the predecessor node throught the pointer in (i - a)-th slot in array, if the color of this node is white, return (i - 1). Otherwise, return Find-Set(i - 1).Predecessor.

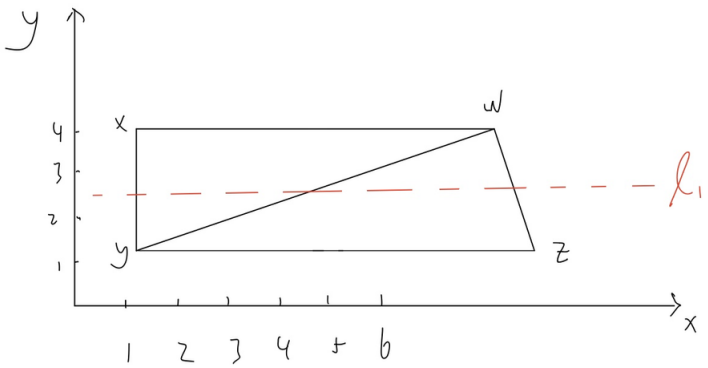
Correctness and Time Complexity:
In general, every time we delete i from S, we check the deletion status for its predecessor and successor, if anyone of them are already deleted, we union them by using union-by-rank and path-compression and update the predecessor attribute of the new root. So when we search the predecessor of i, if the (i-1) still exist in S, we return it. Otherwise, the root of (i-1)'s predecessor is what we want.
Compared to original m times operation of union-by-rank and path compression which cost $\theta(mlog^*m)$, some operations using our algorithm cost constant time and the runtime of Union-by-rank and path compression in our algorithm cost 1 more for updating predecessor of root, so the total run time $\in O(mlog^*m)$. By using aggregate method, for each operation, the cost time $\in O(log^*m)$

Q4. SOLUTION

(a) It does not work. In the following example, this method gives the edges: (x, y) and (y, z). Since the length of (x, z) is obviously smaller than that of (x, y), the actual MST involves (x,z) and (y, z). Thus, this method does not work.



(b) It does not work. In the following example, this method can be divided into two parts by L1 at first, and each part only contains one edge. The light edge between the two parts is (x, y). Thus, MST made by the method involves (x, w), (y, z) and (x, y). Since (w, z) is obviously smaller than (y, z), the weight of the actual MST ((x, w), (x, y), (z, w)) is smaller than the weight of the MST made from the method. Thus, it does not work.



Q5. SOLUTION

(a) Prove by contradiction:

Given a connected undirected weighted graph G, we assume that we have two MST for it: A and B. Firstly, we make a edge set that it contains all edges that only exists either in A or B but not both. Let e1 be the edge with smallest weight in the set. Since weights are distinct, the e1 is a unique edge.

Without loss of generality, we assume e1 is in A. In other words, e1 is not in B. If we add e1 into B, there must be a cycle C in B since B is already connected before adding e1. Since there is no cycle in A, if we compare the cycle C with the corresponding vertices positions in A, there must be an edge e2 which is in cycle C(also in B) but not in A. So e2 is also in the edge set that we create. Then weight of e2 is less than weight of e1.

If we replace e2 with e1 in B, B is still connected. Compared to previous B, new B is smaller weight MST, which is contradiction to our assumption.

Thus, the graph has a unique MST.