DATE ___/___/___

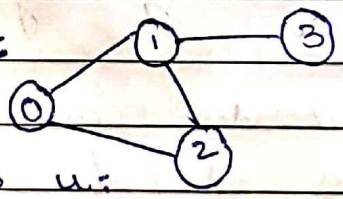
ASSIGNMENT - 4

A1) Given $G_1 = (V, E)$ is a simple connected and undirected graph. T is a tree rooted at u and includes all the nodes of G_1.

Now let us assume $G_1 \neq T$ i.e G_1 cannot be a tree. Since its not a tree, G_1 must contain a cycle, as it is undirected. Let us assume this arbitary cycle as C. Suppose this C contains x nodes $(V_1, V_2 \ldots V_x)$, therefore $V_1 \rightarrow V_2 \rightarrow \ldots \rightarrow V_x \rightarrow V_1$

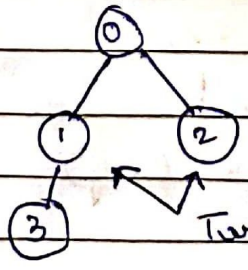
Now if we perform bfs on it, we can observe that since bfs moves level first i.e it searches for all the neighbours and then moves next level, so there will be atleast 2 branches on the bfs tree for the nodes from cycle C.

For example: G_1:



Taking node 0 as u:

T:



Two different branches

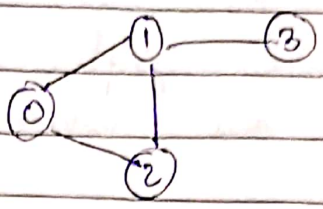
Now on performing dfs on it, nodes present in cycle C will be on same path from root to leaf. Suppose V_i is first visited node, then remaining nodes will be visited at some

Be Positive...

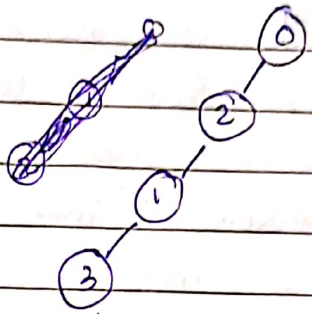
DATE / /

point when v_i is explored, since graph is strongly connected.

For example: G_T :



T :



Hence bfs and dfs produce 2 different trees, which is a contradiction, since both produce trees T. Therefore, G_T is a tree, and since T contains all the nodes and edges of G_T hence $G_T = T$.

Hence proved.

A2) The proof can be written by converting it into the following graph problem.

i) We can consider each unique i,j from pair of specimens to be a node of the graph.

ii) m judgements are undirected edges of the graph between pairs (i,j).

iii) The edges weigh either 1 or 0, 1 if same, 0 if different.

Now we assume A and B as 2 specimens. We take the starting vertex

DATE ___/___/___

and label it A. Then we do a BFS traversal. Between any 2 vertices if the weight of edge is 1 then label of child is same as label of parent, and opposite if weight is 0.

If a node is already visited and we try to ~~as~~ label it is opposite then the graph is inconsistent.

PSEUDOCODE :

```
PROCEDURE isConsistent( ):
    visited[n] ← 0 for all i in range n
    visited[0] ← 1
    q ← initialize a queue with 0.
    while ( q is not empty) :
        curr = q.dequeue()
        for i → 0 to n:
            if adj.get(curr
        for (Edge i : adj.get(curr)):
            if (i.weight != 0):
                if visited[curr] == visited[i.dest] and
                E:
        for (Edge i : adj.get(curr)):
            if ( visited[curr] == visited[i.dest] and
                i.weight == 0 or visited[curr] !=
                visited[i.dest] != 0 and i.dest ==
                    return False
            else if (visited[i.dest] == 0):
                q.enqueue(i.dest)
                if visited[curr] == 1
                    visited[i.dest] = 2
                else
                    visited[i] = 1
```

Be Positive...

DATE ___/___/___

```
if (q.isEmpty):
    for i -> 0 to n:
        if visited[i] == 0:
            q.enqueue(i)
            break
return true
```

We assume theres a data structure Edge with variables source, destination and weight

```
struct Edge {
    int src
    int dest
    int weight
}
```

We have a list adj which contains all the edges attached to a vertex (adjacency list)

A3) This problem can be solved using graphs:

i) We consider each box as a vertex of the graph.

ii) If a box B_1 has a key to another Box B_2 then we consider an directed edge to exist from B_1 to B_2.

iii) We make an adjascency list which contains all the edges from a source, to a destination box.

For example:

n = 4, m = 3

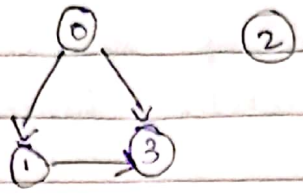
0 has key to 1

0 has key to 3

1 has key to 3

DATE ___/___/___

G (V, E) :



a) We need to do a graph traversal using BFS or DFS from the box chosen by our little brother. If any box remains unvisited after traversal, then it is not possible to open all boxes without smashing one, else it is possible.
We use BFS to traverse:

PSEUDOCODE

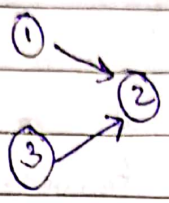
```
PROCEDURE checkUnvisited():
    visited[0 to n] ← false
    key[0 to n] initialized
    bfs(visited, adjlist, start)
    for i → 0 to n:
        if (!visited[i]):
            return false
    return true

PROCEDURE BFS(visited, adjlist, start):
    q ← start          (initialize a queue with start)
    visited[start] ← true
    while (not q.empty):
        curr ← q.dequeue()
        for (Edge i : adjlist.get(curr))
            if (not visited[i.dest]):
                q.enqueue(i.dest)
```

DATE __/__/__

The runtime of bfs is $O(m+n)$ and checking visited is $O(n)$. hence overall $O(m+n)$

b) We simply need to minimize the number of nodes (source) vertices to start bfs from, in order to minimize the number of boxes smashed i.e

eg.

G: 

if we start from ② we need to smash all 3.

But if we start from ① or ③ we need to smash only 2 as ② can be visited from either.

So we need to find the number of parent vertices in a graph, since these vertices do not have an edge pointing 'to' them i.e no key to open them, these are the only boxes we need to smash.

```
PROCEDURE find ( parent [], x ):
    if (parent [x] == -1)
        return x
    return find (parent [], parent [x])

PROCEDURE Union ( parent [], x, y ):
    parentx = find (parent [], x)
    parenty = find ( parent [], y)
```

DATE ___/___/___

parent[parent x] = parent y

```
PROCEDURE numBoxesSmashed():
    parent[0 to n] ← -1,
    for (Edge i : edges):
        parentx ← find(parent, i.src);
        parenty ← find(parent, i.dest);
        if (parentx == parenty)
            continue
        union(parent, parentx, parenty)
    set parent ← parent          (remove duplicates)
    return len(parent)
```