Sirshendu Mandal

160694

Problem 1

The DFS search tree can't have forward or cross edges at any vertex. It is obvious from the definition of unique path graphs. We can't have more than one path from a vertex u to another vertex v in case of an unique path graph. Now

i) forward edge implies => we reach from u to a vertex v by DFS rooted at u and v has already been described before and u is an ancestor v. From definition of unique paths, this can't be allowed as it implies 2 routes)

ii)crossedge implies similarly reaching an already discovered vertex through DFS which doesn't share ancestor / descendant relationships. Similarly for the same reasons as 1.i) ;we can't have cross edges.

So **NO forward / cross edge**

But we need to allow some back edges though to complete some paths.

Criteria for back edges

Two paths composed of back edges can intersect or have in common at most 1 vertex.

Proof

Suppose (u,v) and (x,y) are two paths composed entirely of back edges. Now , say they have in common an edge (m,n) [because here we are making intersection >1..i.e. Atleast 2.,..rest are similarly proved)...now the paths are say x,x1,x2,.....,m,n,p,q,.....y and u,u1,u2,.....,m,n1,n2,.....,v

Since m,n is common region, we obtain a path from x to v as x,x1,x2,.....m,n,n1,n2,.....,v

Again, since its an unique paths graph, there will be atleast some path from x to u say x, x0,x'',.....,u

So we get another path from x to v , i.e. x, x0,x'',....,u,u1,u2,...,m,n,n1,n2,..v

So, we obtain 2 paths from x to v which is in **Contradiction** with unique paths property.

Hence, there can be **ATMOST 2 common nodes** in two back edge paths.

Similarly we can't have a series of back edges which terminate or initiate at same node(we'll get by a procedure similar to above non unique paths then) so..the back edges are atmost O(m/2).... By similar logic , tree edges are O(m/2) also..(tree edges are back edges in reverse, so similar logic and conditions will follow).

So total number of edges |n| is of order |m/2|+|m/2|

Or |n| ~|m|

The O(m^2) algorithm

(i) We will use DFS from every node once. And simultaneously maintain a m x m adjacency matrix which has been default filled with 0s. Now suppose we are invoking DFS from a node u . All the nodes that has been discovered will have their matrix[u][node]=1. Similarly when we will be using DfS from a node , if u is obtained then we will replace matrix[u][node] with 0. (we will be checking along column of [node] now. So overall time complexity is DFs for all nodes + matrix filling. Clearly DFS time is O[m x (m+n)]=O(m^2+mn) and the matrix being manipulated takes O(2m^2) time ~O(m^2) time. Therefore ,

final time complexity is O(m^2+mn) + O(m^2)

~ O(m^2 + mn) **~O[max(m^2,mn)] ……now |n|~|m|**

Therefore ..|mn| is of order m^2 ..or max(m^2,mn)~max(m^2,m^2) =m^2

So we have our **O(m^2) algorithm.**

Pseudo code:

```
Populate matrix[m][m]= all  zeros;
for ( all nodes use DFS)
(use only lower triangular part of the square matrix as rest are symmetrical)
{
DFS(u);
     If node v is discovered && matrix[u][node]==0
         matrix[u][node]=1;
     Else if ( node is discovered && matrix[u][node]==1)
          matrix[u][node]=0;
Shift pointer to next node;
}
If all elements in lower triangular matrix is 1, then unique paths graph
Else
Not a unique paths graph
```

Problem 2

1. The truth table does the job for us. Apart from the T -> F case , allocating any value to a or b doesn't matter . (not a,b)=(T,T).(F,T),(F,F) ..all three have implication's truth value as T. So it's equivalent to (a,b) as (F,T),(T,T),(T,F)..clearly leaving out F,F..therefore not a =>b is equivalent to a OR b.
   Therefore, E=(not x1=>x3)and(not x4=>not x1)and(not x2=>x3)and(not x2=>not x4)
   Clearly, not x1, not x2, x3, not x4 are the vertices and (notx1,x3), (notx4,x1),(notx2,x3),(not x2,not x4) are the directed edges.

2. We will draw all the nodes in a paper (whether they are as usual or in their complement forms). For a=>b , we will draw a directed edge from a to b. Similarly connect all the edges in a network. (i mean all A's or not_A's..keeping in mind that the direction of edge depends upon the implication sign) Now since its CNF..or a collection of ANDs, the whole thing will be true if all the edges are individually satisfiable. Therefore, we can start traversing the graph from any node, and appointing values to them ...keeping in mind that the implication that is represented by the direction of the graph holds true. a=>b and b=>c ..means a=>c ..so by this process if

   #1 we reach a from not_a or not_a from a , then we will have a contradiction, and the CNF wont be satisfiable

   #2 if by if we reach the same vertex and initial point and we are forced use a different valuation as initial, then we will have a contradiction.

3. We will use the idea of strongly connected components. IF in a strongly connected component , both a and not_a exist, then it will be a double implication ..leading to True=False contradiction.
Else for finding a valuation, we need to

a) Find the individual SCCs. //strongly_connected_components
b) We use the condensed form the graph which is directed and acyclic tree.
c) Now we can set T/F all the individual literals inside a single SCC and start assigning values as we deem fit for satisfying the implication..i.e. All pairs other than T implies F is valid. And we have a valuation for the CNF