Theoritical Assignment 4 - ESO207A

Devanshu Somani - 160231

April 2018

1. Problem 1

Part 1 : Answer (1) : Forward Edges

Consider nodes a and b such that b is the descendant of a in DFS tree T_v.

Assume that a forward edge exists from a to b. This will imply that there are 2 paths from a to b.

One : $a \rightarrow b$ (forward edge) and

Second : $a \rightsquigarrow b$ (through the DFS tree T_v's tree edges)

But, it is given that G is a unique path graph and therefore, there cannot exist a and b such that there are more than 1 path to reach b from a. Thus, we have a contradiction.

Thus, our assumption that a forward edge exists is false.

Therefore, **forward edges cannot exist in the DFS tree a unique path graph**.

Answer (2) : Cross Edges

In a single DFS tree T_v, existence of a cross edge $a \rightarrow b$ would mean that there are two paths from the closest common ancestor c (of a and b) and b :

One : $c \rightsquigarrow b$ (The path through tree edges - as c is b's ancestor) and

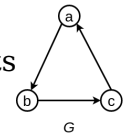
Second : $c \rightsquigarrow a \rightarrow b$ (path though tree edges and then cross edge)

Thus, **a unique path graph cannot have cross edges in a single DFS tree**.

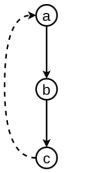
NOTE - Cross edges among two DFS trees of a unique path graph can exist as shown in example : V = {a,b,c} , E={(a,b),(c,b)} where there will be 2 DFS trees with T_a=$a \rightarrow b$ and T_c=c and cross edge $c \rightarrow b$ but still it is a unique path graph.

Answer (3) : Backward Edges

Consider the following graph G and its corresponding DFS tree T_v.
As it is clear from the example that G is a unique path graph and its DFS tree T_a contains a back-edge : $c \rightarrow a$
Thus, **a unique path graph can have back edges in its DFS tree**.



G



DFS Tree T_a

Thus, **for a graph to be a unique path graph it must not have forward edges or cross edges or more than 1 back edge to proper ancestor of a in every subtree rooted at a in a single DFS tree while repeating this by considering each vertex as a source**. The forward and cross edge parts are already proved.

For back edge part, assume that there are more than 2 back edges to proper ancestor of a in a subtree rooted at a. Let 2 of those back edges be $d_1 \rightarrow p_1$ and $d_2 \rightarrow p_2$. Let p_1 be the ancestor of p_2 (otherwise swap them). Now, I claim that there are two paths from a to p_2.

One : $a \rightsquigarrow d_2 \rightarrow p_2$ (as d_2 would be a descendant of a and then the back edge) and

Second : $a \rightsquigarrow d_1 \rightarrow p_1 \rightsquigarrow p_2$ (as p_2 itself is a descendant of p_1).

Therefore, this is the necessary condition for G to be a unique path graph.

Now to prove this condition to be sufficient, let us consider that a graph G does not have forward or cross edges and more than 1 back edge to any proper ancestor of v in any subtree rooted at v in any of its (single) DFS tree and is not a unique path graph. Thus, consider two nodes a and b which have more than 1 path between them. Now there are four cases :

- a **is descendant of** b **in a DFS tree :** One path between them(i.e, $b \rightsquigarrow a$) is through the DFS tree. Now, for the other path from b to a, the path must come through some

cross edge or forward edge from some vertex reachable from b.(as no other tree edge would come to a with connections to b and back edges won't create a path between a node and its descendant). And since, there are no forward or cross edges in the DFS tree, there is only 1 path possible from b to a. Thus, our assumption made above is wrong.

- b **is descendant of** a **in a DFS tree :** Swap a and b from the first case to prove the above assumption wrong.
- a **and** b **are in the same DFS tree but neither of them is an ancestor of the other :** Since, there are no cross edges or forward edges as well. Therefore, the only way to reach a from b (or vice-versa) is though tree and back edges. Let's consider path from a to b (otherwise swap them). Thus, after starting from a, we have to reach the common ancestor of a and b to go from a to b. Back edges would do that work. But in the subtree rooted at a, there can be only atmost 1 back edge to any ancestor of a (assumption). Thus, we can atmost have 1 path from a to b using tree and back edges. Thus, there cannot be more than 1 path from a to b (and vice-versa). Hence, our assumption made above is wrong in this case as well.

Therefore, our assumption that G is not a unique path graph is wrong. Hence, that condition is not only necessary, but is sufficient as well.

Part 2 : Algorithm :

The algorithm would be in such a way :

i. Apply Standard DFS algorithm by taking each unvisited vertex as source.

ii. Check for necessary and sufficient condition that was given in the previous part. Check this condition on every edge of the DFS tree of every SCC of G. Report G to be not a unique path graph upon violation of the condition on any edge (and halt computation).

Pseudo Code :

Forward/Cross Edge condition - (parent.start < node.start) && (visited[node] == true);

Back Edge condition - (parent.start > node.start) && (visited node == true);

```
index = 0;
DFS_Derivative(v,G){
    global_mark[v] = mark[v] = "visited";
    starting_time[v] = index++;
    finishing_time[v] = 0;
    for each w inAdjancencyIist[v]{
        if(mark[w] == "unvisited"){
            DFS_Derivative(v,G);
            for(i=0;i<back_edges(w).size();i++){
                if(finishing_time[back_edges[w][i]]==0)
                    back_edges[v].push_back(back_edges[w][i]);
            }
        }
        else if(global_mark[w]=="unvisited" && mark[w]=="visited"){
            if(starting_time[w] < starting_time[v]){
                if(finishing_time[w]==0){   // Back Edge
                    back_edges[v].push_back(w)
                    if(back_edges[v].size() > 1){
                        print "Not Unique Path Graph"; exit(-1);
                    }
                }
                else{                       // Cross Edge
                    print "Not Unique Path Graph"; exit(-1);
```

```
                    }
                }
                else{                           // Forward Edge
                    print "Not Unique Path Graph"; exit(-1);
                }
            }
        }
        finishing_time[v] = index;
    }
    isUniquePathGraph(G){
        for each v ∈ V{
            if(global_mark[v] == "unvisited") DFS_Derivative(v,G);
            mark[] = {unvisited};
        }
        print "G is a unique path Graph";
    }
```

Time Complexity :

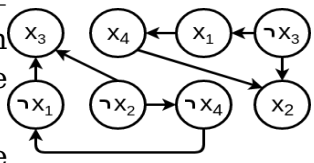
The program exits as soon as forward or cross edge is found. So, it loops only over the back and tree edges. Maximum Tree edges = $|V| - 1$ and for back edges, notice that the back edges can be atmost $O(|V|)$. Proof through induction:

Base Case - $|V| = 0$, back-edges = 0 *Induction Hypothesis* - Let the back-edges be of the order $O(|V|)$ for $|V|$ vertices. *Induction Step* - For $|V| + 1$ vertices, consider subtree of V nodes and one more node. The back-edges thus can be atmost $O(|V|)$(in subtree) + $|V|$(from left out node) = $O(|V| + 1)$ (Now, $|V| = m$ and $|E| = n$). Since only tree edges and back edges and 1 forward/cross edge are traversed through in each DFS in the worst case: Time taken = $\boldsymbol{m \times (O(m) + (m-1) + 1) = m \times O(m) = O(m^2)}$

2. Problem 2

Part 1 :

Each clause in E, i.e, each $a_k = y_i \vee y_j = y_j \vee y_i = (y_i \vee y_j) \wedge (y_j \vee y_i) = (\neg y_i \implies y_j) \wedge (\neg y_i \implies y_j)$ can be represented through 2 edges (both the edges are semantically same). And $\neg y_i$, y_j, $\neg y_j$ and y_i can all be represented though vertices.



For example : $E = (x_1 \vee x_3) \wedge (\neg x_1 \vee x_4) \wedge (x_2 \vee x_3) \wedge (x_2 \vee \neg x_4)$ can be represented as given :-

Thus, the Vertex Set $V = \{\, x, \neg x \mid x \in X \,\}$ and

the Edge Set $\overline{E} = \{\, (\neg u, v), (\neg v, u) \mid (u \vee v)$ is a conjunction in $E \,\}$

i.e, each clause $u \vee v$ represents 2 edges $\neg u \to v$ and $\neg v \to u$

Part 2 : E would not be satisfiable if and only if we are able to find two (or more) clauses in E which are negations of each other (or collectively form a negation) (as in every other case, each one of the clauses/conjunctions can be set to true using a particular assignment of truth values).

Now, in our case all the dependencies (if a then b type of arguments, i.e, $a \implies b$) are denoted by edges, i.e, $a \to b$. Thus, if the dependencies form a cycle among themselves and the negation of some x is seen in the cycle which already has x, then E would not be satisfiable. (as $x \rightsquigarrow \neg x \rightsquigarrow x$ would mean $x \implies \ldots \implies \neg x \implies \ldots \implies x \equiv (x \implies \neg x) \wedge (\neg x \implies x) \equiv x \iff \neg x$ which is a fallacy.

Thus, if there is a path from x to $\neg x$, then there should be no path from $\neg x$ to x. Equivalently, we can say that when the graph is divided in to its strongly connected components, then for E to be satisfiable, there should not be any component which has both x and $\neg x$ inside it.

This condition is not only sufficient but also is necessary because for a disjunction of conjunctions to be non-satifiable, it is necessary that atleast two of its disjunctions become

negations of each other (followed by theory of logic). And this is exactly prevented from happening in that condition.

Part 3 : Convert E into a graph as shown in Part 1 above. Now, if E is satisfiable, then x and $\neg x$ could not be in a same cycle or in a same strongly connected component, to be accurate.

Now break the graph into its strongly connected components and follow these steps :

i. Initially, set the current component as the one with the highest finishing time and assign each of the vertex inside that component to True and the negations of all the vertices inside it to be False.

ii. Consider the next component which has the highest finishing time of the remaining components.
   - If any of its vertex is already assigned to some value say k (can be True or False), then assign k to all the vertices of that component.
   - If none of its vertices are assigned, then assign each of its vertices as True and the negations of all the vertices inside it to be False.

iii. Repeat Step ii until all the vertices of the graph are assigned some value.

This assignment of values is possible because it is given that E is satisfiable and thus, from Part 2, it is known that to vertices which are negations of each other cannot be in the same strongly connected component and thus all vertices could be set to the same value. Also, it will be evaluate E as true. The proof is given below :

Proof - Assume the contrary. That E is satisfiable and still this assignment of values is evaluating E to be false.

Thus, there should be atleast 1 edge in the graph corresponding to E which is of the form $u \to v$ and the assignment is of u is True and that of v is False. Let $u \to v$ be the first such violation as seen in decreasing order of finishing times, i.e, No edge of the form $a \to b$ exists if a is assigned True, b is assigned False and finishing time of a is greater than that of v.

Now, consider the DFS trees after implementing Strongly Connected Components Algorithm(i.e, each DFS tree would represent 1 strognly connected component of vertices) of the graph corresponding to E. There are 2 cases for the edge $u \to v$:

- **The edge is completely inside a strongly connected component**, i.e, u and v must both belong to the same strongly connected component. This is false because we are assigning each node in any strongly connected component with the same value and thus no such u and v can be found with values True and False respectively.
- **The edge is across two different strongly connected components**, i.e, u and v are in different strongly connected components. Now, in the graph with each DFS tree representing a strongly connected component, it is known that the an edge cannot go from a component having smaller finishing time to a component having a larger finishing time (this is the basis of the SCC algorithm). Thus u must belong to a component with a larger finishing time and v must belong to a component with a smaller finishing time. Now, by our definition of the vertex and edge sets, if edge $u \to v$ exists, then $\neg v \to \neg u$ also exists. Also, v is assigned False, this can only be possible if $\neg v$ is assigned True before that implying finishing time of $\neg v$ is larger than that of v. Now, we made the assumption above that $a \to b$ can't exist with a being True, b being False and finishing time of a being greater than v. Now, since $\neg v \to \neg u$ exists, thus, $\neg u$ has to be assigned True. But we already assumed that u is assigned True and our algorithm does not allow any u and $\neg u$ both to be assigned True as E is satisfiable and u and $\neg u$ are not in a strongly connected component. Therefore, its a contradiction.

Thus, in both the possible cases, we arrive at a contradiction. This implies that our assumption that such edge $u \to v$ could exist is False. Thus no such edge $a \to b$ exists where a is assigned True and b is assigned False by this algorithm.

Hence, this algorithm is correct to produce a truth value assignment which satisfies E.