Data Structures and Algorithms
Assignment 4 Solutions

Ayush Bansal
Roll No. 160177

I Problem 1 Solution

1.1 Part 1

We are given a **unique path graph** G and we have to comment on the existence of forward, cross and backward edges in the DFS tree T_v of a vertex v.

Forward Edge | Cross Edge | Backward Edge | 2 Backward Edges

Forward Edges
In case of forward edges in the graph, consider the example in the figure above, there are 2 paths from vertex A to C and thus forward edges are **Not Possible**.

Cross Edges
In case of cross edges in the graph, consider the example in the figure above, there are 2 paths from vertex A to C and thus cross edges are **Not Possible**.

Backward Edges
In case of backward edges, consider the example in the figure above, there are no 2 different paths for any pair of vertices, thus cycles are allowed in the directed graph as backward edges are allowed. So, backward edges are **Possible and Allowed**.

2 Backward Edges from a single node
In case of 2 backward edges from a single node in the graph, consider the example in the figure above, there are 2 paths from vertex C to B and thus 2 backward edges from a single node are **Not Possible**.
Thus, the graph can atmost have $V - 1$ back edges (leaving out root vertex).

Lemma 1.1. *The necessary and sufficient condition for a directed graph G to be a unique path graph is "there shouldn't be any forward or cross edges in the DFS tree T_v of any vertex $v \in V$".*

Proof. Let's assume G is a unique path graph and there is a forward or a cross edge in certain DFS tree T_v for some $v \in V$.
From the above discussion on the the existence of forward, cross and backward edges in the DFS tree of vertex T_v, it is clear that if there is a forward or a cross edge, then the path from a vertex A to a vertex C s.t. $A, C \in V$ will not be unique and thus the graph G will not be a unique path graph which is a **contradiction**.
Thus, for each $v \in V$, there shouldn't be any forward or cross edge in the DFS tree T_v otherwise, the uniqueness of atleast one path will be lost and the graph G will not be a unique path graph.

Now, Let's assume that G is not unique path graph and has no forward or cross edges in any DFS tree T_v for all $v \in V$.
Since G is not a unique path graph, there exist nodes $a, b \in V$ such that there are 2 different paths from vertex a to vertex b. If there are 2 paths from vertex a to b then there will be a DFS tree T_v for some $v \in V$ which will contain a cross or forward edge from vertex a to b leading to **contradiction**. Thus, the graph G must be a unique path graph. □

1.2 Part 2

Since any DFS tree T_v of the graph cannot have more than $V - 1$ back edges and also the tree edges will be equal to $V - 1$, the total number of edges in a unique path graph $E \leq 2V - 2$.
I will perform a **DFS traversal** taking each node of the graph as source one at a time, if in any DFS tree T_v, a forward or a cross edge is found, then the graph is not a unique path graph.
Following will be the algorithm of **modified DFS** for detecting forward and cross edges.

Algorithm 1 DFS Traversal

```
procedure DFS(u)                          ▷ Takes source vertex, color is global array
    color[u] ←GREY                         ▷ Grey means vertex discovered
    for all v ∈ Adj[u] do
        if color[v] =BLACK then            ▷ If forward or cross edge is found
            flag ← false                   ▷ Flag is global Boolean
            return
        else if color[v] =WHITE then       ▷ Move to adjacent undiscovered vertex
            DFS(v)
        end if
    end for
    color[u] ←BLACK                        ▷ Black means Vertex has finished traversal
end procedure
```

Below will be the algorithm which will call the above procedure and report if the graph is unique path graph or not.

Algorithm 2 UniqueOrNot

```
procedure UNIQUEORNOT(Graph G)            ▷ Returns true if Unique path graph
    flag ← true                            ▷ flag will be returned finally
    for all u ∈ V[G] do                    ▷ Calling DFS for each vertex
        if flag = false then
            break
        end if
        for all v ∈ V[G] do                ▷ Mark all vertices undiscovered for new dfs
            color[v] ←WHITE                ▷ White means vertex undiscovered
        end for
        DFS(u)                             ▷ Calling DFS procedure with u as root
    end for
    return flag                 ▷ Flag is set to false within DFS if not unique path graph
end procedure
```

Time Complexity

The time complexity of **DFS** is $O(V + E)$ but is reduced to $O(V)$ since $E \leq 2V - 2$ (already shown) otherwise we will definitely find a forward edge and algorithm will end.
The time complexity of **UniqueOrNot** procedure will be $O(V(V + E))$ and since $E \leq 2V - 2$, the final complexity will be $O(V^2)$.

II Problem 2 Solution

We are given a set of N boolean variables $X = \{x_1, \dots, x_N\}$ and an expression $E = a_1 \wedge a_2 \wedge \dots a_K$ s.t. $a_k = y_i \vee y_j$ where y_i, y_j are literals.
For the question,

$$E = (x_1 \vee x_3) \wedge (\neg x_1 \vee x_4) \wedge (x_2 \vee x_3) \wedge (x_2 \vee \neg x_4)$$

2.1 Part 1

We know that $a \vee b \equiv b \vee a$ and $a \vee b \equiv \neg a \Rightarrow b$, so we will have

$$a \vee b \equiv (\neg a \Rightarrow b) \wedge (\neg b \Rightarrow a)$$

Thus, the converted expression E can be written in 2 forms both of which will be equivalent (one is extended version) as $a \vee b \equiv b \vee a$.

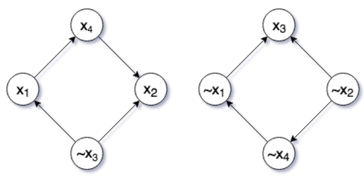
$$E = (\neg x_1 \Rightarrow x_3) \wedge (x_1 \Rightarrow x_4) \wedge (\neg x_2 \Rightarrow x_3) \wedge (\neg x_2 \Rightarrow \neg x_4)$$

$$E = (\neg x_1 \Rightarrow x_3) \wedge (\neg x_3 \Rightarrow x_1) \wedge (x_1 \Rightarrow x_4) \wedge (\neg x_4 \Rightarrow \neg x_1) \wedge (\neg x_2 \Rightarrow x_3) \wedge (\neg x_3 \Rightarrow x_2) \wedge (\neg x_2 \Rightarrow \neg x_4) \wedge (x_4 \Rightarrow x_2)$$

For constructing the graph we will choose **boolean variables and their negations** as the Vertex set and the **Implication relations** as the Edge set.

- 2 edges for every clause i.e. $\neg A \rightarrow B$ and $\neg B \rightarrow A$ for clause $A \vee B$.
- 1 node for every Boolean variable involed in the Boolean expression i.e. A, $\neg A$, B, $\neg B$ etc.
- So, total $2K$ edges for K distinct clauses and $2N$ vertices for N boolean variables.

Following is the graph for the expression E, it will have $2N = 8$ vertices and $2K = 8$ edges.



2.2 Part 2

Now, we have got a graph from the expression E and we have to find whether E is satisfiable or not.

Lemma 2.1. *If both x_i and $\neg x_i$ lie in the same **SCC** (Strongly Connected Component), the CNF (Expression) is **unsatisfiable**, otherwise it is **satisfiable**.*

Proof. Consider the following cases of existence of edges which obey the rules of implication (if A then B).

1. Edge $x_i \Rightarrow \neg x_i$ exists, then x_i must be **false** otherwise the statement can't be satisfiable.
2. Edge $\neg x_i \Rightarrow x_i$ exists, then x_i must be **true** otherwise the statement can't be satisfiable.
3. Both edges $\neg x_i \Rightarrow x_i$ and $x_i \Rightarrow \neg x_i$ exist, then the statement is **unsatisfiable**.

Using the fact that a path (SCC) like $a \Rightarrow b \Rightarrow c \Rightarrow a$ in graph will have same truth values for all.
If there is a path both from x_i to $\neg x_i$ and from $\neg x_i$ to x_i, then the expression is **unsatisfiable**.
The above statement is same as saying that x_i and $\neg x_i$ are part of the same **SCC** (Strongly Connected Component) as there exists a path from x_i to $\neg x_i$ as well as $\neg x_i$ to x_i.
If x_i and $\neg x_i$ do not lie in the same SCC, then it is possible to assign truth values to the variables according to **case 1 & 2** above and thus, the expression is **satisfiable**. ☐

Algorithm to find Satisfiably:

- Find all the **SCCs** in the graph using the algorithm studied in class.
- Check if any SCC contains both a boolean variable and its negation i.e. x_i and $\neg x_i$.
  - If yes (i.e. it contains both), then report **unsatisfiable**.
  - Else, report **satisfiable**.

In case of graph of expression E, there is no SCC containing both x_i and $\neg x_i$, thus it is **satisfiable**.

2.3 Part 3

We are given a **satisfiable** expression E and I have to assign truth values to the variables in X.

Lemma 2.2. *The graph formed by merging all vertices in an **SCC** (Strongly Connected Component) into a single vertex and doing it for all SCCs is a **Directed Acyclic Graph**.*

Proof. Assume graph formed by mentioned process is not a **DAG**, then there is a cycle containing atleast 2 vertices.
Since these vertices represent 2 different SCCs and there is a path leading from SCC1 to SCC2 and vice versa, thus there will be a path from each and every vertex of SCC1 to SCC2 and also from SCC2 to SCC1 and thus, they will not remain different SCCs anymore which is a contradiction to our original graph.
Thus, the graph formed must be a Directed Acyclic Graph. □

Given an expression E, the complete algorithm for assigning truth values to the variables in X is

- Build a directed graph using each **SCC** as a vertex and the edges from this SCC to others as edges of this graph, basically merging all the vertices in an SCC into one.
- Perform a **topological sort** on the above newly formed graph (already proved it is a **DAG**).
- Assign truth values to each **SCC** in order of topological sort **till all the boolean variables have an assigned truth value** as follows.
  - If all the boolean variables in an SCC are unassigned, assign all the truth value of **TRUE** and their negations the truth value of **FALSE**.
  - If a boolean variable in SCC is assigned some value, then assign the same value to the rest of the variables in that SCC and give the opposite value to negations.
  - Stop when all the boolean variables in $X = \{x_1, \ldots, x_n\}$ are assigned.

Using the above algorithm, there could be 2 possible orderings of the SCC graph formed by expression E, so their corresponding truth values will be:

Topological - Ordering	**Truth Values**
$\neg x_3, x_1, x_4, x_2, \neg x_2, \neg x_4, \neg x_1, x_3$	x_3 -> False x_1, x_2, x_4 -> True
$\neg x_2, \neg x_4, \neg x_1, x_3, \neg x_3, x_1, x_4, x_2$	x_3 -> True x_1, x_2, x_4 -> False

Taking example of **first** ordering:

- Assign $\neg x_3$ -> True, and x_3 -> False.
- Assign x_1 -> True, and $\neg x_1$ -> False.
- Assign x_4 -> True, and $\neg x_4$ -> False.
- Assign x_2 -> True, and $\neg x_2$ -> False.
- Stop as all variables have been assigned.