COL351: Analysis and Design of Algorithms **Fall 2024**

Tutorial Sheet 4 Solutions

Announced on: Aug 15 (Thurs)

Problems marked with ($\star$) will not be asked in the tutorial quiz.

1. Consider running the Topological-Sort-via-DFS algorithm on a directed graph G that is not acyclic. The algorithm will not compute a topological ordering (as none exists). Does it compute an ordering that minimizes the number of edges that travel backward? Justify your answer.

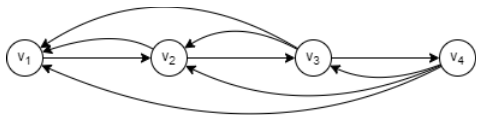

Figure 1: Counter Example for Q1

No, the algorithm does not compute an ordering that minimizes the number of backward edges. For example, consider the graph G of Figure 1.
In G, if the Topological-Sort-via-DFS algorithm starts from v_1, then $S = [v_1, v_2, v_3, v_4]$ is a possible ordering output by the algorithm. It can be seen this ordering has 6 backward edges. However, the ordering $O = [v_4, v_3, v_2, v_1]$ only has 3 backward edges and hence, the ordering S does not minimise the number of backward edges.
Additional Reading: See Minimum Feedback Arc Set problem. How does this relate to the problem of finding an ordering that minimises number of backward edges?

2. The game of Snakes and Ladders has a board with n cells where you seek to travel from cell 1 to cell n. To move, a player throws a six-sided dice to determine how many cells forward they move. The board also contains snakes and ladders that connect certain pairs of cells. A player who lands on the mouth of a snake immediately falls back down to the cell at the other end. A player who lands on the base of a ladder immediately travels up to the cell at the top of the ladder. Suppose you have rigged the dice to give you full control of the number for each roll. Design an efficient algorithm to find the minimum number of dice throws to win.

Assumption: For simplicity, we assume that there is no *chain* of snakes and ladders in the board, i.e., if one ladder/snake takes us from cell i to j, then there is no ladder/snake starting at cell j.
Informal Idea: Construct a graph $G = (V, E)$ where the vertices correspond to the cells of the board and the edges going out of a vertex v correspond to the possible cells we can reach from v in one move.

Note: We will be using the notation ladder to refer to both ladders and snakes, as a snake can just be thought of as a downward ladder.
The algorithm is as follows:

ALGORITHM 1: Snakes and Ladders

```
Initialize the directed graph $G = (V, E)$ with vertices $V = \{v_1, v_2, \ldots, v_n\}$ and $E = \emptyset$
/* The vertex $v_i$ corresponds to the $i^{th}$ cell of the board */
foreach $v_i \in V$ do
    for $j \in \{i+1, \ldots, i+6\}$ and $j \leqslant n$ do
        if some ladder starts at $j$ and ends at at $k$ then
            Add the edge $(v_i, v_k)$ to $E$
        else
            Add the edge $(v_i, v_j)$ to $E$
/* Edges have been added to $G$. Now find shortest path from $v_1$ to $v_n$ */
Output shortest path length from $v_1$ to $v_n$ using BFS from $v_1$
```

Proof of Correctness: Let us define a *valid* sequence as a sequence of dice throws, which gets us from 1 to n in the Snakes and Ladders game. By definition, we want to find out a *valid* sequence of minimum possible length. Now, by simulating the dice throws of a valid sequence on our graph G, the following claim is straightforward:
Claim 1. *Every* valid *sequence S has a corresponding path P from v_1 to v_n. Similarly, every corresponding path P from v_1 to v_n has a* valid *sequence S. Also, the length of S is equal to the number of edges in path P.*
From the above claim, we get that the shortest path length from v_1 to v_n is equal to the smallest number of dice throws required to go from 1 to n.
Time Complexity: Since there are n vertices in G and every vertex has at most 6 outgoing edges, $|E| = O(n)$. So, adding the edges to G takes $O(n)$ time and BFS also takes $O(n)$ time. Hence, overall algorithm takes $O(n)$ time.
Do it yourself: Where have we used the assumption we made at the start? What happens if we do not make this assumption, can we still obtain a linear time algorithm?

3. Let u and v be two vertices in a directed graph $G = (V, E)$. Design a linear-time algorithm to find the number of distinct shortest paths (not necessarily vertex disjoint) between u and v.

Informal Idea: Start BFS from node u and keep track of distances from u to all other nodes and the number of distinct shortest paths to each node from u. As BFS explores the graph, it updates the shortest distance for each node and adds to the path count if that node can be reached via the same shortest path length.

ALGORITHM 2: Count the Number of Distinct Shortest Paths from u to v

```
Input: Graph $G = (V, E)$, vertices $u, v \in V$
Output: Number of distinct shortest paths from $u$ to $v$
foreach $x \in V$ do
    $dist[x] \leftarrow \infty$
    $count[x] \leftarrow 0$
end
$dist[u] \leftarrow 0$
$count[u] \leftarrow 1$
Initialize a queue $Q$
$Q$.enqueue($u$)
while $Q$ is not empty do
    $x \leftarrow Q$.dequeue()
    foreach neighbor $y$ of $x$ do
        if $dist[y] = \infty$ then
            $dist[y] \leftarrow dist[x] + 1$
            $count[y] \leftarrow count[x]$
            $Q$.enqueue($y$)
        end
        else if $dist[y] = dist[x] + 1$ then
            $count[y] \leftarrow count[y] + count[x]$
        end
    end
end
return $count[v]$
```

Proof of Path Counting using Induction:
We want to show that after running BFS, count$[v]$ correctly represents the number of distinct shortest paths from vertex u to any vertex v.
Base Case (Distance 0):
The BFS starts from u, and initially, dist$[u] = 0$ and count$[u] = 1$ because there is exactly one way to reach u (starting at u itself). For all other vertices $v \neq u$, count$[v] = 0$ because no paths have been discovered to them yet.
This establishes the base case: at distance 0, the path count is correct.
Inductive Hypothesis:
Assume that for every vertex x with dist$[x] \leqslant d$, count$[x]$ correctly represents the number of distinct shortest paths from u to x.
Inductive Step:
We need to show that the path count is correct for vertices at distance $d + 1$.

a) **Exploring Neighbors:**
Consider a vertex y such that dist$[y] = d + 1$. By the properties of BFS, y is reached from some vertex x with dist$[x] = d$. According to the algorithm, when y is reached for the first time:

$$\text{dist}[y] \leftarrow \text{dist}[x] + 1 \quad \text{and} \quad \text{count}[y] \leftarrow \text{count}[x].$$

This update is correct because all shortest paths to y at distance $d + 1$ must pass through some vertex x at distance d.

b) **Counting Multiple Paths:**
If y is encountered again from another vertex x' with $\text{dist}[x'] = d$, then:

$$\text{count}[y] \leftarrow \text{count}[y] + \text{count}[x'].$$

This update correctly accumulates the number of distinct shortest paths to y because any shortest path to y must come from one of its predecessors at distance d. By the inductive hypothesis, the path counts $\text{count}[x]$ and $\text{count}[x']$ are already correct, so adding them gives the correct number of paths to y.

- Written "I do not know how to approach this problem" - 0.6 points
-
  - Correct algorithm idea - 2 points
  - Proof by induction idea, i.e., induction can be done on distance of the vertex from u - 1 point

4. A directed graph $G = (V, E)$ is *weakly connected* if for every pair of vertices (u, v), there is a path from u to v or from v to u or both ways. Design a linear time algorithm to check if G is weakly connected.

Notation: Let $G' = (V', E')$ denote the condensation graph of a graph $G = (V, E)$, $SCC(v)$ denote the vertex in V' corresponding to strongly connected component of $v \in V$ and $ord(v')$ denote the index in the topological order of graph $G' = (V', E')$ where $v' \in V'$.

Informal Idea: The algorithm to check if a directed graph is weakly connected involves first finding its strongly connected components (SCCs). Each SCC is a subgraph where every vertex is reachable from every other vertex within that subgraph. We then create a condensation graph where each SCC is represented as a single vertex, and edges exist between these vertices if there is a connection between their corresponding components in the original graph. Since the condensation graph is a directed acyclic graph (DAG), we check for any two consecutive vertices (u', v') in the topological order of the condensation graph whether $(u', v') \in E'$. If it does, the original graph G is weakly connected otherwise not.
Algorithm:

a) Find the Strongly Connected Components (SCCs) and construct Condensation graph G' of G using Kosaraju's algorithm.

b) Create a topological ordering of the condensation graph.

c) Check whether for any two consecutive vertices (u', v') in the topological order of the condensation graph whether $(u', v') \in E'$, if it is then Graph G is weakly connected otherwise not.

Claim: For all pair of consecutive vertices (u', v') in the topological order of the condensation graph, $(u', v') \in E'$ if and only if G is weakly connected.
Proof: ($\Rightarrow$) For all pair of consecutive vertices (u', v') in the topological order of the condensation graph, $(u', v') \in E' \implies$ there exists a DFS tree of condensation graph G' which is a straight line. Following two cases are possible for the pair of vertices (u, v) where $u, v \in V$:

a) $SCC(u) \neq SCC(v)$: WLOG assume $ord(SCC(u)) < ord(SCC(v))$, then there exists a path from u to v because the dfs tree is a straight line. Hence (u, v) is weakly connected.

b) $SCC(u) = SCC(v)$: Since u and v belong to the same SCC. There exists a path from u to v and vice versa. Hence (u, v) is weakly connected.

($\Leftarrow$) The condensation graph of a weakly connected graph is also weakly connected because for any pair of vertices (u, v) such that $SCC(u) \neq SCC(v)$ are weakly connected i.e there is path from u to v or vice versa $\implies$ there is a path from $SCC(u)$ to $SCC(v)$ or vice versa. As we know that condensation graph G' is a DAG and weakly connected graph $\implies$ for any two consecutive vertices in the topological order of G' contains an edge, if not then G' is not weakly connected.

- Written "I do not know how to approach this problem" - 0.6 points
-
  - Forming the condensation graph - 0.5 points
  - Topological sort - 0.5 points
  - In the condensation graph, checking whether every consecutive vertex pair has an edge b/w them - 0.5 points
  - High-level proof of correctness - 1 point
  - Claiming $O(m+n)$ time complexity - 0.5 points

5. A *vertex cover* of an undirected graph $G = (V, E)$ is a subset of vertices $U \subseteq V$ such that each edge in E is incident to at least one vertex of U. Design an efficient algorithm (using BFS or DFS) to find a minimum-size vertex cover if G is a tree.

Hint: What can we say about the leaves of the tree? Do we need them in a minimum-size vertex cover?
Informal Idea: The key observation is the following: Let v be any leaf of the tree and let u be its parent (observe that a leaf node always exists). Then, if there is an optimal

vertex cover, say S, such that $v \in S$, then there is another optimal vertex cover S' such that $v \notin S'$ and $u \in S'$. That is, without loss of generality, we can assume that an optimal vertex cover does not contain any leaf node. This makes intuitive sense: By including the leaf node in the vertex cover, we *only* cover the parent-leaf edge, whereas by including the parent node, we cover the parent-leaf edge as well as possibly other edges.

This argument suggests the following recursive algorithm: Identify all leaf nodes of a tree (using BFS or DFS). For any edge $\{u, v\}$ where v is a leaf, add u to the vertex cover and remove the $\{u, v\}$ edge from the graph. After this step, we are left with a forest. We can ignore all single-node trees because they are already covered by the current solution. For each remaining tree (strictly smaller in size) in the forest, we repeat the same algorithm.

Algorithm: The algorithm presented in the informal idea can be performed in linear time in the following manner: Consider an arbitrary vertex r as root of the tree. Then, for each child c of r, we recursively find the minimum vertex cover of the subtree rooted at vertex c and check if c is present in this vertex cover. If not, we add r to the vertex cover. Note that recursively solving for all the children first is equivalent to performing a DFS traversal of the tree, and adding the vertex r to vertex cover is equivalent to deleting r from G.

ALGORITHM 3: Minimum Vertex Cover for a Tree

```
Input: Tree $T$ with root $r$
Output: Minimum Vertex Cover of the tree
1 Initialise set $S$ of vertices as $\emptyset$
2 Function FindMinimumVertexCover($T$, $r$):
3     Function VC($v$):
          /* Set of vertices to be added to the vertex cover */
4         foreach child $c$ of $v$ do
5             VC( $c$ )
6         foreach child $c$ of $v$ do
7             if $c \notin S$ then
8                 Add $r$ to $S$
9     return $S$
```

Proof of Correctness: The following claim is central to our algorithm:
Claim 2. *If v is a leaf of the tree, then there exists a minimum-sized vertex cover S such that $v \notin S$ and $parent(v) \in S$.*

Proof. We prove it *by contradiction*. Suppose for a leaf vertex v, there does not exist any minimum vertex cover S such that $v \notin S$ and $parent(v) \in S$. Then, consider any minimum vertex cover O. By our assumption, we know that $v \in O$ or $parent(v) \notin O$. Note that $O' = O \setminus \{v\} \cup \{parent(v)\}$ is also a vertex cover of G, with size no more than O and hence, O' is also a minimum vertex cover containing $parent(v)$ but not v.

Hence, we arrive at a contradiction. □

For a vertex $u \in V$, suppose u is part of some minimum vertex cover of G. Also, consider a graph $G^{'}$, obtained by deleting u from G and let $O^{'}$ be a minimum vertex cover of $G^{'}$. Then, we can claim the following:
Claim 3. *$O^{'} \cup \{u\}$ is a minimum vertex cover of G.*

Proof. Try it yourself (Can be proved easily using *contradiction*). □

From the above two claims, we see that the algorithm is correct.

Time Complexity: Note that in a tree, the vertices of the subtrees rooted at children of the root are disjoint. So, we can maintain a global array, which stores whether a vertex has been included in our solution till some point of the algorithm, then checking whether membership queries like $c \notin S$ and "adding" a vertex to S take only $O(1)$ time. So, we spend only $O(deg(v))$ time at any vertex v and hence, our algorithm takes $O(|E|)$ time. Since $|E| = n - 1$ for trees, the algorithm runtime is $O(n)$.

6. An *independent set* of an undirected graph $G = (V, E)$ is a subset of vertices $U \subseteq V$ such that no edge in E is incident to two vertices of U. Design an efficient algorithm (using BFS or DFS) to find a maximum-size independent set if G is a tree.

Hint: What can we say about the leaves of the tree? Do we need them in a maximum-size independent set?
Informal Idea: This question is very similar to the previous question. Here, we can prove that there exists a maximum independent set containing all the leaves of the tree if number of vertices of the tree is at least 3. If number of vertices becomes less than 3, then we can handle that case separately.

Algorithm Sketch: Consider an arbitrary vertex r as root of the tree. Then, for each child c of r, we recursively find the maximum independent set of the subtree rooted at vertex c and check if c is present in this independent set. If all children c are not present in their respective independent sets, we add r to the independent set. Note that recursively solving for all the children first is equivalent to performing a DFS traversal of the tree.
Proof of Correctness: The proof is along the lines of the previous question, with some modifications. Try it yourself.

7. Given an undirected graph G, an *independent vertex cover* of G is a subset of vertices that is both an independent set and a vertex cover of G. Design an efficient algorithm for testing whether G contains an independent vertex cover.

Informal Idea: Check whether graph G is bipartite or not.
Algorithm: Refer to Problem 6 of Tutorial 3. It takes $O(m+n)$ time.
Claim: Undirected graph G contains an independent vertex cover if and only if G is bipartite.
Proof: ($\Rightarrow$) The graph G contains an independent vertex cover means that there exists a subset of vertices $U \subseteq V$ such that for every edge $(u,v) \in E$ either $u \in U$ or $v \in U$. Thus we can partition V into two sets $V_1 = U$ and $V_2 = V \setminus U$, then $V_1 \cap V_2 = \emptyset$, $V_1 \cup V_2 = V$ and there are no edges in the same set which means G is bipartite.
($\Leftarrow$) Assume that $G = (V, E)$ is a bipartite graph, with the vertex set V partitioned into two disjoint sets V_1 and V_2 such that every edge in E has one endpoint in V_1 and the other in V_2.
We will show that either V_1 or V_2 forms an independent vertex cover.

a) **Vertex Cover Condition:** Consider the set V_1. By the construction of the bipartite graph, every edge in E is incident to at least one vertex in V_1 (or similarly in V_2). Hence, V_1 is a vertex cover of G. The same is true for V_2.

b) **Independence Condition:** Since G is bipartite and V_1 and V_2 are disjoint, there are no edges between any two vertices within V_1, and similarly, no edges within V_2. Therefore, both V_1 and V_2 are independent sets.

Thus, if G is bipartite, we can pick either V_1 or V_2 as an independent vertex cover.

Note: No marks will be deducted for assuming that a linear-time algorithm exists for checking whether G is bipartite, as this has already been discussed in Tutorials.

- Written ”I do not know how to approach this problem” - 0.6 points
-   - Claiming that *independent vertex cover* exists if and only if G is bipartite - 1 point
    - Proof idea (forward implication) - 1 point
    - Proof idea (backward implication) - 0.5 points
    - Algorithm for testing bipartiteness (Even just mentioning that the algorithm has been discussed previously is enough) - 0.5 points

8. ($\star$) Given an undirected graph $G = (V, E)$, a vertex $v \in V$ is said to be an *articulation point* if $G \setminus \{v\}$ has more connected components than G. Design an $\mathcal{O}(|V| + |E|)$ algorithm to find all articulation points of G.

Informal Idea: Pick an arbitrary vertex of the graph *root* and run depth first search from it. Note the following fact:

- Let's say we are in the DFS, looking through the edges starting from vertex $v \neq root$. If the current edge (v, to) is such that none of the vertices to or its descendants in the DFS traversal tree has a back-edge to any of ancestors of v, then v is an articulation point. Otherwise, v is not an articulation point.
- Let's consider the remaining case of $v = root$. This vertex will be the point of articulation if and only if this vertex has more than one child in the DFS tree.

Algorithm: Now we have to check the above fact for each vertex efficiently. We'll use "time of entry into node" computed by the depth first search.
So, let $tin[v]$ denote entry time for node v. We introduce an array $low[v]$ which will let us check the fact for each vertex v. $low[v]$ is the minimum of $tin[v]$, the entry times $tin[p]$ for each node p that is connected to node v via a back-edge (v, p) and the values of $low[to]$ for each vertex to which is a direct descendant of v in the DFS tree:

$$low[v] = \min \begin{cases} tin[v] & \\ tin[p] & \text{for all } p \text{ for which } (v,p) \text{ is a back edge} \\ low[to] & \text{for all } to \text{ for which } (v,to) \text{ is a tree edge} \end{cases}$$

Now, there is a back edge from vertex v or one of its descendants to one of its ancestors if and only if vertex v has a child to for which $low[to] < tin[v]$. If $low[to] = tin[v]$, the back edge comes directly to v, otherwise it comes to one of the ancestors of v.
Thus, the vertex v in the DFS tree is an articulation point if and only if $low[to] \geqslant tin[v]$.

Claim: If vertex $v \neq root$ is a articulation point then there is no descendant of v in the DFS tree which has a back-edge to any of ancestors of v.
Proof by Contradiction:
Assume v (where $v \neq$ root) is an articulation point, but suppose for contradiction that there exists a descendant u of v in the DFS tree that has a back-edge to an ancestor a of v. If such a back-edge exists, then even if v is removed, the subtree rooted at u can still connect to the rest of the graph through this back-edge to a. This would mean that the removal of v does not increase the number of connected components, contradicting the definition of v as an articulation point. Therefore, no such back-edge can exist if v is indeed an articulation point.

9. ($\star$) In the 2SAT problem, you are given a set of clauses, each of which is the disjunction (logical "OR") of two literals. (A literal is a Boolean variable or the negation of a Boolean variable.) You would like to assign a value TRUE or FALSE to each of the variables so that all the clauses are satisfied, with at least one true literal in each clause. For example, if the input contains the three clauses $x_1 \vee x_2$, $\neg x_1 \vee x_3$, and $\neg x_2 \vee \neg x_3$, then one way to satisfy all of them is to set x_1 and x_3 to "TRUE" and x_2 to "FALSE".

   Design an algorithm that determines whether or not a given 2SAT instance has at least one satisfying assignment. (Your algorithm is responsible only for deciding whether or not a satisfying assignment exists; it need not exhibit such an assignment.) Your algorithm should run in $\mathcal{O}(m+n)$ time, where m and n are the number of clauses and variables, respectively.

[Hint: Show how to solve the problem by computing the strongly connected components of a suitably defined directed graph.]

Informal Idea: Consider any clause $C_i = x \vee y$, where x, y are literals. If x is `F`, then y will necessarily need to be `T` for C_i to be `T`. Similarly, if y is `F`, then x will necessarily need to be `T`. This can be captured by writing each clause $x \vee y$ as $(\neg x \Rightarrow y) \wedge (\neg y \Rightarrow x)$ (you can derive this expression using boolean rules also). This is known as implication normal form of the expression (INF).
Note that as soon as we set x (or y) to `T`, all the clauses containing occurrences of $x, \neg x$ (or $y, \neg y$) will also change. So, to capture the effect of these implications over all clauses, we construct a directed graph of these implications, also known as implication graph. For each literal x and $\neg x$ we will have a corresponding vertex v_x and $v_{\neg x}$. The edges will correspond to the implications present in the implication normal form (INF) of our 2-SAT expression. For example, if INF contains the expression $x \implies y$, then we add a directed edge from v_x to v_y in G. This edge effectively denotes that if we set x to `T`, then we "must" set y to `T` in a satisfying assignment.
In this implication graph, for any literal x, there is a path from x to $\neg x$ and from $\neg x$ to x if and only if there is no satisfying assignment.

Algorithm:
Assume that the boolean variables in the CNF are $x_1, x_2, \ldots, x_n$ and the clauses are $C_1, C_2, \ldots, C_m$. Then, the algorithm is the follo wing:

- Initialise a graph $G = (V, E)$ with $V = \{v_{x_1}, v_{\neg x_1}, v_{x_2}, v_{\neg x_2}, \ldots, v_{x_n}, v_{\neg x_n}\}$ and $E = \emptyset$.
- For $i = 1$ to m:
  - Let x_j and x_k be the literals such that $C_i = (x_j \vee x_k)$. Add the edges $(v_{\neg x_j}, v_{x_k})$, $(v_{\neg x_k}, v_{x_j})$ to G.
- Find the SCCs of G (using Kosaraju's Algorithm).
- For $i = 1$ to n: Check if the vertices $v_{x_i}, v_{\neg x_i}$ lie in the same connected component. If yes, then no satisfying assignment possible. If $v_{x_i}, v_{\neg x_i}$ lie in different SCC's for all i, then a satisying assignment is possible.

Proof Of Correctness: We prove the following claim:
Claim 4. *There exists a literal x such that x and $\neg x$ lie in the same SCC if and only if the 2SAT expression is unsatisfiable.*

Proof. ($\implies$) We will prove this by contradiction, suppose the 2SAT expression is satisfiable. Since there exists a path P from v_x to $v_{\neg x}$ (assume P contains the edges $(v_x, v_{y_1}), (v_{y_1}, v_{y_2}), \ldots, (v_{y_k}, v_{\neg x})$, where $y_i's$ are literals), the implications $(x \implies y_1), (y_1 \implies y_2), \ldots, (y_k \implies \neg x)$ are present in the INF. From these, we can infer that $x \implies \neg x$. Since there is also a path from $v_{\neg x}$ to v_x in G, we can similarly infer that $\neg x \implies x$. So, in a satisfying assignment of the 2SAT expression, a literal x is assigned "`TRUE`" if and only if $\neg x$ is also "`TRUE`", which is not possible and hence, leads to a contradiction.

$(\impliedby)$ We show that if v_x and $v_{\neg x}$ lie in different SCCs for all literals x, then there exists a satisfying assignment to 2SAT. First, note the following observation, which is straightforward from our construction of G:

Observation 1. For 2 literals x, y, if an edge (v_x, v_y) is present in G, then the edge $(v_{\neg y}, v_{\neg x})$ is also present in G.

Let's denote the SCC's of G by $SCC_1, SCC_2, \ldots, SCC_k$. WLOG, we also assume that these SCC's are in the topological order, i.e., for all i, j such that $i < j$, no edge goes from SCC_j to SCC_i. As discussed in the lectures, this ordering always exists.
For a literal x, we denote the index of the SCC in which v_x lies as $SCC(x)$. It is given to us that $SCC(x) \neq SCC(\neg x)$
Now, consider the following boolean assignment A: If $SCC(x) < SCC(\neg x)$ in the ordering, then $A(x) = \mathtt{F}$, else $A(x) = \mathtt{T}$.
We now show *by contradiction* that the above assignment A is a satisfying assignment for 2SAT. Assume that A is not a satisfying assignment. Then, in the INF, there exists an implication $(x \implies y)$ whose value is `F` according to the assignment A.

- This occurs only if $A(x) = \mathtt{T}, A(y) = \mathtt{F}$. This implies that $SCC(\neg x) < SCC(x)$ and $SCC(y) < SCC(\neg y)$.
- As $(x \implies y)$ exists in INF, there is an edge (v_x, v_y) in G. Hence, $SCC(x) \leqslant SCC(y)$.

Combining the above two claims, we get that $SCC(\neg x) < SCC(\neg y)$.
By Observation 1, there also exists an edge $(v_{\neg y}, v_{\neg x})$ in G. So, $SCC(\neg y) \leqslant SCC(\neg x)$, which is a contradiction.
Hence, the assignment A is a satisfying assignment.

$\square$

Time Complexity: Since the graph G has $2n$ vertices and $2m$ edges, Kosaraju's Algorithm takes $O(m + n)$ time and checking if $(v_{x_i}, v_{\neg x_i})$ lie in the same SCC for all i takes $O(n)$ time. Hence, the overall algorithm runs in $O(m + n)$ time.