Theory Assignment-2: ADA Winter-2023

Deepanshu Dabas (2021249) Ankush Gupta (2021232)

---

1. The police department in the city of Computopia has made all the streets one-way. But the mayor of the city still claims that it is possible to legally drive from one intersection to any other intersection.
(a) Formulate this problem as a graph theoretic problem and explain why it can be solved in linear time. (7 Marks)
(b) Suppose it was found that the mayor's claim was wrong. She has now made a weaker claim: "if you start driving from town-hall, navigating one-way streets, then no matter where you reach, there is always a way to drive legally back to the town-hall. Formulate this weaker property as a graph theoretic problem and explain how it can be solved in linear time. (8 Marks)

Ans:
Part a.
Formulating Problem into A Graph Theoretic Problem

We can convert this problem into graph theoretic problem by following:

A directed graph $G = (V, E)$, where V is the set of vertices representing intersections in the city of Computopia, and E is the set of directed edges representing the one-way streets between the intersections, the problem is to determine whether the graph G is strongly connected (Mayor of the city claims that it is possible to legally drive from one intersection to any other intersection, i.e. there is a way from each vertex to another vertex which implies from every vertex u to every other vertex v, there exists a path that implies the graph is strongly connected graph).

We interpret each directed edge $(u, v) \in E$ as a one-way street from intersection u to intersection v i.e. it is legal to drive from u to v, but not from v to u.

Use of graph traversal algorithms

Since after formulating the problem into a graph theoretic problem, we need to determine whether the graph is strongly connected or not. If the mayor claim is true, then the graph will be strongly connected and only have 1 component. We will use ***Kosaraju's Algorithm*** since this algorithm is a linear-time algorithm that can determine whether a directed graph is strongly connected.

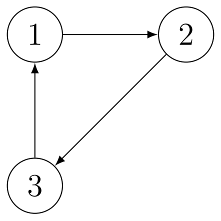
Here is how we will use it to solve the problem:

Algorithm: Kosaraju's Variation

1. Perform a depth-first search (DFS) of G and keep track of the order in which the vertices are visited. Let this order be denoted by L. Specifically, whenever we finish visiting a vertex, we add it to the front of a linked list.
2. Reverse the direction of all edges in G to obtain G_{rev}.
3. Perform another DFS of G_{rev}, starting from the vertex at the front of the linked list L. Each DFS tree in this step corresponds to a strongly connected component of G.
4. **If there is only one DFS tree, then G is strongly connected and we return true. Otherwise, G is not strongly connected and we return false.**

Example:

Consider the directed graph G = (V, E) with V = 1, 2, 3 (Street Endpoints) and E = (1, 2), (2, 3), (3, 1). E represents intersection roads.



We can see that there exists a directed path from every vertex u in V to every other vertex v in V, since there is a cycle in the graph. Therefore, the graph is strongly connected and hence claim is also true which is case if we verify manually.

To illustrate how Kosaraju's algorithm works for this example, we can perform the following steps:

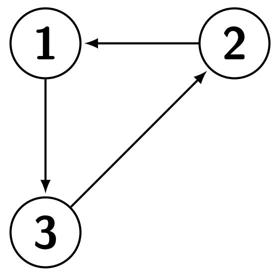
Step 1: Perform a depth-first search of G and keep track of the order in which the vertices are visited. Let this order be denoted by L. Specifically, whenever we finish visiting a vertex, we add it to the front of a linked list.

Starting from vertex 1, the DFS traversal order is: 1, 2, 3.

Therefore, L = 3, 2, 1.

Step 2: Reverse the direction of all edges in G to obtain G_rev.

The resulting graph $G_{rev} = (V, E_{rev})$ has V = 1, 2, 3 and E_{rev} = (2, 1), (3, 2), (1, 3).



Step 3: Perform another DFS of G_{rev}, starting from the vertex at the front of the linked list L. Each DFS tree in this step corresponds to a strongly connected component of G.

Starting from vertex 3, the DFS traversal order is: 3, 2,1.

The DFS tree rooted at vertex 3 corresponds to the strongly connected component 3, 2,1.

Since there is only one DFS tree, G is strongly connected and we return **true**.

Necessary And Sufficient Conditions

For the problem of determining whether Mayor claim is true or not we need to determine whether G is strongly connected or not, the necessary and sufficient condition is that there exists a path from every vertex u to every other vertex v, for (u,v) $\in$ G.

This means that for every pair of vertices (u, v) $\in$ G, there must be a directed path from u to v and a directed path from v to u.

The Kosaraju's algorithm works by identifying the strongly connected components of the directed graph G. It is based on the fact that if G is strongly connected, then for every vertex $u, v \in$ G, there exists a path from u to v and a path from v to u and hence algorithm will work in this problem.

Therefore, the necessary and sufficient condition for the correctness of *algorithm* is that it correctly identifies all the strongly connected components of G. This implies that if *Kosaraju's algorithm* returns a single strongly connected component, then G is strongly connected and Mayor claim is *true*, and if it returns multiple strongly connected components, then G is not strongly connected and hence mayor claim is *false*.

Time Complexity Analysis

Following Steps Are Involved in algorithm:

1. Performing a DFS on G to obtain the order L in which the vertices are visited $\rightarrow \mathcal{O}(|V| + |E|)$
2. Reversing the direction of all edges in G to obtain $G_{\text{rev}} \rightarrow \mathcal{O}(|E|)$

3. Performing a DFS on G_{rev} starting from the vertex at the front of L and checking whether there is only one DFS tree.$\rightarrow \mathcal{O}(|V| + |E|)$

(We used fact that time complexity of DFS on graph is $\rightarrow \mathcal{O}(|V| + |E|)$)

From 1), 2) and 3), we get,

Time complexity of algorithm $\rightarrow \mathcal{O}(|V| + |E|) + \mathcal{O}(|E|) + \mathcal{O}(|V| + |E|) = \mathcal{O}(|V| + |E|)$

Here $|V| \rightarrow$ Number of vertices in graph and $|E| \rightarrow$ Number of edges in graph.

Since the time complexity of the algorithm is linear as expected(Kosaraju's algorithm time complexity is linear) ,this problem can be solved in linear time.

Part b.

Formulating Problem into A Graph Theoretic

We can formulate this problem similar to part a. Let T be the vertix representing the town hall in the directed graph $G = (V, E)$, where V is the set of vertices and E is the set of edges in the graph. For each directed edge $(u, v) \in E$, we can assume that there is a one-way street from vertex u to vertex v.

Constructing a graph in the same way as in the first part of the question this problem is equivalent to asking, whether the townhall is a vertix in a sink strongly connected component: If the townhall is in a sink SCC, then all the vertices reachable from the townhall are in the same scc as the townhall, so by the definition of strongly conncted components there is a path back to the townhall from all the vertices reachable from the townhall. If the townhall is not in a sink SCC, then we can reach a vertex w from the townhall that is not in the same SCC as the townhall, and then we will not find a path from w back to the townhall .

This problem can be formulated as a graph theoretic problem of finding whether T is a vertex in a sink strongly connected component of G.

To construct the graph, we can represent it using an adjacency list. For each vertex $u \in V$, we store a list of all vertices $v \in V$ such that there is a one-way street from u to v. The adjacency list representation of G is denoted as $\mathit{AdjLst}(G) = (u, \mathit{Adj}(u)) \mid u \in V$, where $\mathit{Adj}(u)$ is the list of vertices that are adjacent to u in G.

In summary, the problem can be stated as follows:

Given a directed graph $G = (V, E)$ with adjacency list representation $\mathit{AdjLst}(G)$, and a vertex $T \in V$ representing the town hall, determine whether T is a vertex in a sink strongly connected component of G.

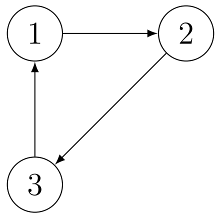
Use of Graph Traversal Algorithms

The algorithm checks whether the townhall is in a sink SCC: We find the strongly connected components of the graph. Mark all the vertices that are in the same SCC as the townhall. Then we run DFS starting from the townhall until we get stuck. We will check whether all the vertices reachable from the townhall are in the same SCC as the townhall. If that is the case the mayor is right. If the set of vertices reachable from the townhall contains a vertix which is not in the same scc as the townhall the opposition is right.

Algorithm 1 Check if there exists a path from every vertex to the townhall

```
Require: A directed graph $G = (V, E)$ and a vertex $t$ representing the townhall
Ensure: True if there exists a path from every vertex to the townhall, false otherwise
 1: function CheckPathToTownhall($G, t$)
 2:     $SCCs \leftarrow$ Kosaraju($G$)              ▷ Find strongly connected components
 3:     $marked \leftarrow$            ▷ Mark all vertices in the same SCC as the townhall
 4:     for $SCC \in SCCs$ do
 5:         if $t \in SCC$ then
 6:             $marked \leftarrow marked \cup SCC$
 7:     $visited \leftarrow$ DFS($G, t$)             ▷ Run DFS starting from the townhall
 8:     for $u \in visited$ do
 9:         if $u \notin marked$ then
10:             return False                                ▷ "Opposition is right"
11:     return True                                         ▷ "Mayor is right"
```

Example
Consider the directed graph G = (V, E) with V = 1, 2, 3 (Street Endpoints) and E = (1, 2), (2, 3), (3, 1). E represents intersection roads, town-hall is V = 1.



Steps: SCCs = [2, 3, 1] marked = 2, 3, 1 visited = 1, 2, 3
In this case, the output will be true ("Mayor is right") since all vertices reachable from the townhall (vertix 1) are in the same strongly connected component as the townhall.
Necessary And Sufficient Conditions
The algorithm is correct because it relies on the following necessary and sufficient conditions:

1. If the townhall is in a sink strongly connected component, then all vertices reachable from the townhall are also in the same strongly connected component. Because if the townhall is in a sink strongly connected component, then there is

a path from the townhall to every other vertix in the same strongly connected component.

2. If the townhall is not in a sink strongly connected component, then at least one vertix reachable from the townhall is not in the same strongly connected component as the townhall.
3. Because If the townhall is not in a sink strongly connected component, then there is a path from the townhall to a vertix in a different strongly connected component. Since there is a path from the townhall to this vertix, it is reachable from the townhall.

We first finds the strongly connected components of the graph, and marks all vertices that are in the same strongly connected component as the townhall. Then, it runs a DFS starting from the townhall and checks if all vertices reachable from the townhall are also marked. If they are, then the townhall is in a sink strongly connected component and the mayor is right. Otherwise, the opposition is right.

The correctness of the algorithm follows directly from the necessary and sufficient conditions described above.

Running Time Analysis
The time complexity of the algorithm can be broken down into the following steps:

1. Constructing the adjacency list: $O(|V| + |E|)$
2. Running the Kosaraju's algorithm to find SCCs: $O(|V| + |E|)$
3. Marking all vertices in the same SCC as the townhall: $O(|V|)$
4. Running DFS starting from the townhall: $\mathcal{O}(|V| + |E|)$
5. Checking if all reachable vertices are in the same SCC as townhall: $\mathcal{O}(|V|)$

Therefore, the overall running time of the algorithm is $\mathcal{O}(|V| + |E|)$.

$$\mathcal{O}(|V| + |E|) + \mathcal{O}(|V| + |E|) + \mathcal{O}(|V|) + \mathcal{O}(|V| + |E|) + \mathcal{O}(|V|) = \mathcal{O}(|V| + |E|)$$

2. Given an edge-weighted connected undirected graph G = (V, E) with n + 20 edges. Design an algorithm that runs in O(n)-time and outputs an edge with smallest weight contained in a cycle of G. You must give a justification why your algorithm works correctly. (15 Marks) (Hint: Formulation into graph theoretic problem is not necessary as the graph is given already.)

Ans:
Formulating Problem into A Graph Theoretic Problem:

Formulation into graph this theoretic problem is not necessary as the graph is given already

Use of graph traversal algorithms
We will use algorithm based on application of DFS to find all the cut-edges (Tarjan's Algorithm). *This algorithm was taught in the tutorial.*
An edge is a cut edge if and only if not contained in any cycle. Therefore if we mark all cut edges, then the rest of the edges will be part of at least one of the cycles and hence we will need to find minimum weight of these edges i.e. the minimum weight of all edges contained in a cycle of G.
We will create a boolean map initialized to true by default for every value. We will mark all cut-edges as false using Tarjan's Algorithm(application of DFS algorithm) in this boolean map.

Tarjan's Algo:
The algorithm is a modified version of depth-first search (DFS) that assigns a unique "depth" value to each node in the graph and uses this information to identify cut edges.

Algorithm to find minimum weight edge:

1. Applying Tarjans Algo
    (a) Start with an unvisited vertex v and initialize its discovery time to 0, and a variable time to 0.
    (b) Mark the vertex v as visited and push it onto a stack.
    (c) Increment the time variable by 1 and set the discovery time of v to time.
    (d) For each neighbor w of v:
    a. If w is not visited, recursively call the algorithm on w.
    i. Set the parent of w to v and update its low-link value to be the minimum of its own discovery time and the low-link value of its neighbors.
    ii. If the discovery time of w is less than the low-link value of v, (v,w) is a cut edge. Mark it as false.
    iii. If the low-link value of w is greater than or equal to the discovery time of v, pop all vertices from the stack starting from w until v, and add them to the list of cut edges.

    b. If w is visited and not the parent of v, update the low-link value of v to be the minimum of its current value and the discovery time of w.

2. For each edge $v = (u, w) \in E$:
   if $isCycle(v)$ is not false:
   $smallest\_weight = min(smallest\_weight, weight[v])$
3. return smallest_weight

Algorithm 2 Edge with smallest weight contained in a cycle of G.

```
1: function TARJAN(G,isCycle)
2:     Start with an unvisited vertex $v$ and initialize its discovery time to 0, and a variable time to 0.
3:     Mark the vertex $v$ as visited and push it onto a stack.
4:     Increment the time variable by 1 and set the discovery time of $v$ to time.
5:     for each neighbor $w$ of $v$: do
6:         if $w$ is not visited then
7:             Recursively call the algorithm on $w$.
8:             Set the parent of $w$ to $v$ and update its low-link value to be the minimum of its own discovery time and the low-link value of its neighbors.
9:             if the discovery time of $w$ is less than the low-link value of $v$, $(v, w)$ is a cut edge. Mark $isCycle[v][w]$ as false. then
10:            if the low-link value of $w$ is greater than or equal to the discovery time of $v$ then
11:                Pop all vertices from the stack starting from $w$ until $v$, and add them to the list of cut edges.
12:        else
13:            if $w$ is visited and not the parent of $v$ then
14:                Update the low-link value of $v$ to be the minimum of its current value and the discovery time of $w$.
15: function MINIMUMWEIGHT(G)
16:
Require: Edge-weighted connected undirected graph G = (V, E) with n + 20 edges.)
17:
18:     $isCycle \leftarrow$ boolean map of edges which maps every edge [u][v] $\rightarrow$ true , for (u,v) $\in$ E
19:     Tarjan(G,$isCycle$)
20:     $smallest\_weight \leftarrow \infty$
21:     for each edge $(u, v)$ in $G$: do
22:         if $isCycle[u][v]$ = true then
23:             $smallest\_weight$ = min($smallest\_weight$,weight$(u, v)$)
24:     $print(smallest\_weight)$
25:     return $smallest\_weight$
```

The necessary and sufficient conditions that explains why algorithm works correctly.

G is an edge-weighted connected undirected graph with n + 20 edges.

1) Claim: Tarjan's algorithm correctly identifies all the cut edges in G.

Proof: Tarjan's algorithm identifies all the cut edges by performing a depth-first search on G and maintaining a low-link value for each vertex. A cut edge is an edge that, if removed, would result in a disconnected graph.

In the algorithm, if the discovery time of w is less than the low-link value of v, then (v, w) is a cut edge.
Conversely, if the low-link value of w is greater than or equal to the discovery time of v, then the vertices on the stack between v and w form a strongly connected component and (v, w) is not a cut edge.

Therefore, the algorithm correctly identifies all the cut edges in G.

2) Claim: The function minimumweight correctly identifies the smallest weight edge contained in a cycle of G.

Proof: The function minimumweight iterates over all edges in G and checks whether each edge is in a cycle by using the boolean array isCycle that is set by Tarjan's algorithm. If the edge is in a cycle, its weight is compared to the current minimum weight. At the end of the iteration, the function returns the smallest weight edge contained in a cycle of G.

By the definition of a cycle, a cycle is a closed path in G that contains at least one edge.

Therefore, if there is a cycle in G, there must be at least one edge in the cycle.
Since the algorithm correctly identifies all the cut edges in G, any edge that is not a cut edge must be in a cycle.

Hence, the function minimumweight correctly identifies the smallest weight edge contained in a cycle of G.

Therefore, the algorithm works correctly, and above mentioned are necessary and sufficient conditions that explain why the algorithm works correctly.

Running Time Analysis

1. The Tarjan's algorithm which is used in to find all cut edges visits each vertex and edge of the graph exactly once.
   We can neglect the number of vertices (v) in the time complexity analysis as the algorithm only needs to visit each vertex once during the DFS traversal.
   In the worst case, the DFS traversal can visit all the vertices, but the number of edges in the graph is always greater than or equal to the number of vertices minus one (i.e., E >= V - 1), since a connected graph with fewer than V - 1 edges is not possible.

   Therefore, the time complexity of Tarjan's algorithm can be expressed as $\mathcal{O}(N)$, where N >= V - 1 and number of edges $= N + 20$
2. The second part iterating over all edges and checking minimum weight is $\mathcal{O}(n)$ as it will iterates over all $n + 20$ edges and all other arithmetic operations are $\mathcal{O}(1)$
3. Initialization of various data structures such as map will be $\mathcal{O}(N)$ at worst.

Therefore, the overall time complexity of the algorithm is $\mathcal{O}(N)+\mathcal{O}(N)+\mathcal{O}(N)+\mathcal{O}(1) = \mathcal{O}(N)$.
Here N=n and n is given in question

3. Suppose that G be a directed acyclic graph with following features .
• G has a single source s and several sinks t1, . . . , tk .

• Each edge (v → w) (i.e. an edge directed from v to w) has an associated weight p(v → w) between 0 and 1.
• For each non-sink vertex v, the total weight of all the edges leaving v is
∑ (v→w)∈E Pr(v → w) = 1.

The weights Pr(v → w) define a random walk in G from the source s to some sink ti ; after reaching any non-sink vertex v, the walk follows the edge v → w with probability Pr(v → w). All the probabilities are mutually independent. Describe and analyze an algorithm to compute the probability that this random walk reaches sink ti

for every i ∈ 1, . . . , k. You can assume that an

arithmetic operation takes O(1)-time. (see Figure 1 for an illustration) (20 Marks)

Ans: Dynamic Programming-Based Approach

Preprocessing stage

We need to find the topological order of the graph (Can be done using any algorithm such as Kahn's algorithm).
Subproblem definition

Let $Pr(v, T_i)$ be the probability that the random walk starting from vertex V reaches sink T_i.
The value of $Pr(v, T_i)$ will be a real number between 0 and 1, where a value of 0 indicates that the walk starting from v cannot reach T_i, and a value of 1 indicates that the walk will always reach T_i.

Recurrence Relation

The recurrence for computing the probability $Pr(v, t_i)$ can be written as follows:

$$Pr(v, t_i) = \begin{cases} 1 & \text{if } v = sourcevertex \\ 0 & \text{if } v \text{ is a sink and } v \neq t_i \\ \sum_{(v \to w) \in E} Pr(w, t_i) \cdot p(v \to w) & \text{otherwise} \end{cases}$$

where:

v and t_i are vertices in the graph.
E is the set of edges in the graph.
$(v \to w)$ is an edge from vertex v to vertex w.

$p(v \to w)$ is the probability of transitioning from vertex v to vertex w.

This recurrence computes the probability that the random walk starting from vertex v reaches the sink t_i. The first case handles the base case, where the starting vertex is the source vertex, so the probability is 1. The second case handles the base case where the starting vertex is a sink but is not t_i, so the probability is 0. The third case is the recursive case, where we sum over all edges $(v \to w)$ and compute the probability of transitioning from v to w and then reaching the sink ti from w.

Pseudocode / Description of Algorithm

We will use topological sort-based algorithm for computing probabilities in a directed acyclic graph (DAG) ,based on the principle of dynamic programming, where the probability of reaching a vertex v is calculated based on the probabilities of reaching its parent vertices.

The algorithm starts by computing a topological ordering of the vertices in the DAG. This ensures that when processing a vertex v, all its parent vertices have already been processed. The algorithm initializes the probability of the starting vertex to be 1 and the probability of all other vertices to be 0.

For each vertex v in the topological order, the algorithm processes all its outgoing edges (v, w) and updates the probability of reaching vertex w as follows:
$\Pr[w] = \Pr[w] + \text{prob}[v] \times p(v \to w)$.
Here, $p(v \to w)$ is the probability of transitioning from vertex v to vertex w.

By the end of the algorithm, the probability of reaching each vertex in the DAG from the starting vertex is computed and stored in the array Pr.
Function will take *input* of Graph, edge list with weights, vertices list .
Correctness of Algorithm

The correctness of the algorithm can be proved by induction on the vertices in the topological order.

Base Case: For the starting vertex, its probability is initialized to 1, which is correct.

Inductive Hypothesis: Assume that the algorithm correctly computes the probabilities of reaching all vertices that come before the vertex v in the topological order.

Inductive Step: Consider a vertex v in the topological order. By the inductive hypothesis, the probabilities of reaching all its parent vertices have been computed correctly. For each outgoing edge (v, w), the algorithm updates the probability of reaching vertex w as $\Pr[w] = \Pr[w] + \Pr[v] \times p(v \to w)$. By the inductive hypothesis, the probability of reaching vertex v is correct. Therefore, the probability of reaching vertex w is also correct.

Thus, by induction, the algorithm correctly computes the probabilities of reaching all vertices in the DAG from the starting vertex.

```
function ComputeProbability($G$)
Require: A directed graph $G = (V, E)$ ($E$ contains weighted edges)
    $n \leftarrow$ Size($V$)
    $topo\_order \leftarrow$ TopologicalSort($G$)
    Pr $\leftarrow$ array of size $n$, initialized to 0
    $s \leftarrow$ index of starting vertex in $V$
    Pr[$s$] $\leftarrow$ 1
    for $v$ in $topo\_order$ do
        if $G[v]$ is empty then
            continue
            // Since this vertex will be sink vertex as there are no edges to process for this vertex

        // Each edge (v $\rightarrow$ w) (i.e. an edge directed from v to w) has an associated weight p(v $\rightarrow$ w) between 0 and 1.
        for $(w, p)$ in G[v] do
            Pr[$w$] $\leftarrow$ Pr[$w$] + Pr[$v$] $\times$ $p$
    result $\leftarrow$ empty array initialized to 0 // To store result of the sinks

    for $i$ in $t$ do // t is the array that have all the sink vertices
        result[$i$] $\leftarrow$ prob[i]
    return result
```

Running Time Analysis

The running time analysis of this algorithm can be broken down into the following steps:

1. Computing the input size and arithmetic operation takes takes constant time: $\mathcal{O}(1)$.
2. Topological sorting of the graph, which takes $\mathcal{O}(|V| + |E|)$ time using algorithms such as depth-first search (DFS) or Kahn's algorithm.
3. Initializing the probability array(Pr) take $\mathcal{O}(|V|)$ time.
4. Processing each vertex in the topological order and updating its probability takes $\mathcal{O}(|V| + |E|)$ time.
5. Constructing and returning the result array takes $\mathcal{O}(|V|)$ time.

Therefore, the overall time complexity of this algorithm is $\mathcal{O}(|V| + |E|)$
$(\mathcal{O}(|V| + |E|) + \mathcal{O}(|V|) + \mathcal{O}(|V| + |E|) + \mathcal{O}(|V|) = \mathcal{O}(|V| + |E|))$

Acknowledgement:

1. Tutorial- 7,8
2. Lecture Slides Of Strongly Connected Component,Directed DFS, DP - Maximum Weight Independent Set of a Tree
3. CodeHelp YT Channel