ADA Theory Assignment - 2

Aakarsh Jain
Roll Number - 2021507
IIIT - Delhi
aakarsh21507@iiitd.ac.in

Siddhant Rai Viksit
Roll Number - 2021565
IIIT - Delhi
siddhant21565@iiitd.ac.in

April 08, 2023

1 Question 1

1.1 Problem

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.

- Formulate this problem as a graph theoretic problem and explain why it can be solved in linear time.
- 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.

1.2 Formulating to Graph problem

It is assumed that we are provided the input of intersections, and streets joining these intersections. Let's assume that there is n instersections and m edges.

We can then start by labelling the town-hall as 0, and label the remaining $n-1$ intersections from 1 to $n-1$. We can also similarly label the streets from 0 to $m-1$. Now, we can think of each intersection as a vertex in the graph and the one-way streets as edges, directed in direction of the street. We will thus create set V for all the vertices and set E to contain all the edges. Thus, we have obtained a directed graph G using E and V.

We will now create an adjacency list adj. If there is a street from intersection u to v, then we will add v to $adj(u)$. Similarly, we will also create an reversed adjacency list rev_adj for part (b), i.e., If there is a street from intersection u to v, we will add u to $rev\_adj(v)$

1.3 Approach

1.3.1 Part a

We will use the Kosaraju's algorithm to distribute the graph G into its Strongly Connected Components (SCC). Then we will check if all the vertices in set V belong to the same SCC. If yes, then Mayor's claim is true, otherwise it is false.

1.3.2 Part b

We will first mark all the vertices reachable from the source using DFS as part of set A. We will then reverse all the edges and mark all vertices reachable now as the set B. If $A \subset B$, the claim is proved, otherwise it is false.

1.4 Kosaraju's Algorithm to find SCC

We can use Kosaraju's Algorithm to find the SCC for each vertex of the graph as discussed in Lecture-12 of the course.

The pseudo-code for the same can be found here, or a detailed implementation in C++ can be found here along with basic proof of correctness.

Here we will assume that the function *kosaraju* implements the above algorithm, and takes an adjacency list as input and returns a vector *scc* with $scc[u]$ denoting the SCC of vertex u.

1.5 Proof of Correctness

1.5.1 Part a

According to mayor's claim, for any 2 intersection, there must be a path from one to another. In the graph, this translates such that for any 2 vertices u and v, there must be a path from u to v and vice versa. Thus, by definition, u and v must belong to the same SCC. Since this is true for any 2 vertices, the entire set V must belong to the same SCC, i.e., the graph G has a single SCC containing all the vertices.

To verify the same, we will use the Kosaraju's algorithm as described above to find the SCC's and check if all the vertices belong to the same SCC.

1.5.2 Part b

Let us denote $P(u \to v, G)$ to mean there exists a path from u to v in the graph G. Let us denote $R(G)$ of a graph G as the graph with the same vertices as G but with all edges reversed. Now,

$$P(v \to u, G) \to P(u \to v, R(G)) \tag{1}$$

The mayor's claim is
$P(u \to v, G) \to P(v \to u, G) \forall v$.
Using (1)
$P(u \to v, G) \to P(u \to v, R(G)) \forall v$
We can construct the set $v | P(u \to v, G)$ and $v | P(u \to v, R(G))$ using DFS and compare them.

1.6 Pseudo-Code

1.6.1 part a

```
1: procedure checkClaim(adj)
2:     n ← length(adj)
3:     scc ← kosaraju(adj)
4:     for i from 0 to n − 1 do
5:         if scc[i] ≠ scc[i + 1] then
6:             return False
7:         end if
8:     end for
9:     return True
10: end procedure
```

1.6.2 part b

```
procedure DFS(u,visited, adj)
    visited[u] ← 1
    for v in adj[u] do
        if visited[v] ≠ 1 then DFS(v)
        end if
    end for
end procedure
procedure Solve(adj,revadj,v,townhall)
```

```
    A ← [0, .., 0]
    B ← [0, ..., 0]
    DFS(townhall,A,adj)
    DFS(townhall,B,revadj)
    for u in [1, .., v] do
        if A[u] = 1 and B[u] = 0 then return False
        end if
    end for
return True
end procedure
```

1.7 Time Complexity

1.7.1 Part a

The time complexity for Kosaraju's algorithm is $O(|V| + |E|)$, and then we check if all the vertices of the graph belong to the same SCC in $O(V)$, since we require $O(1)$ time to check the SCC any vertex belongs to.

Thus, the total Time Complexity is $O(|V| + |E|)$.

1.7.2 Part b

The time complexity for the two DFS calls is $O(|V| + |E|)$, and comparing A and B is $O(|V|)$. Preprocessing *adj* and *revadj* is $O(|E|)$. Thus our final time complexity will be $O(|V| + |E|)$

2 Question 2

2.1 Problem

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.

2.2 Formulating to Graph problem

We are already provided a graph with n vertices and $n + 20$ edges. We can denote the set of vertices of graph as V and set of edges as E. We will thus store the graph using an adjacency list *adj*, where $adj(u)$ is the set of all vertices v such that there exists an edge between u and v. We will also use the function $getWeights(u, v)$, which returns the list of weights of all the edges between u and v.

2.3 Approach

Since the provided graph is an undirected graph, if an edge does not belong to a cycle, then it is a bridge (removing the edge from the graph will cause the number of connected components to increase by 1), and vice versa.

Thus, we can use a simple DFS - Traversal using the algorithm to find bridges described below to find the edges which belong to a cycle, and then find the least weighted edge among all such edges.

NOTE: The above algorithm assumes that there are no self loops in the graph, i.e, there is no edge which has same endpoints.

2.4 Finding Bridges

The algorithm for finding bridges is very similar to that of finding cut-vertices already described in tutorials. We will use a DFS traversal to visit every node in the graph. For the purpose of this algorithm, we can start DFS from any node. We will then compute the following:

We will store the start time of DFS Traversal for every node in the array *tin*. We will also initialize *tin* as -1 to indicate that the vertex has not been visited yet in the DFS Traversal.

We can now define *low* for our DFS tree as:

$$low(u) = min \left\{ \begin{array}{ll} tin(u) & \\ tin(p) & \forall p \text{ such that} (u,p) \text{ is a back edge} \\ low(v) & \forall v \text{ such that} (u,v) \text{ is a forward edge} \end{array} \right\} \quad (2)$$

Thus, for an edge (u, v) in the DFS tree, it is a bridge iff $low(v) > tin(u)$

NOTE that a DFS tree is a directed tree, where edges indicate the direction of DFS traversal.

2.5 Proof of Correctness

We need to prove that in a weighted undirected graph G, if an edge a is not a bridge, then a must belong to a cycle in graph G.

We will prove the above using contradiction, Let's assume that there exists an edge a in graph G, which is not a bridge and does not belong to a cycle in graph G. Since a is a bridge, there must exist a path P in graph G which connects the vertices u and v and does not contain the edge a. But, if we create a path P' by adding edge a to path P, then P' forms a cycle. This is a contradiction to our assumption, and thus by way of contradiction, our claim is correct.

We can also prove that if an edge a belonging to undirected graph G is a bridge, then a cannot belong to a cycle.

We will again use method of contradiction and assume there exists and edge b which is a bridge and belongs to a cycle C. Now, let u and v be 2 vertices belonging to C. As u and v belong to a cycle, there must be 2 disjoint paths from u to v. Thus, if we remove the edge a from the graph, u and v still remain connected using the other path. Thus, the edge a cannot be a bridge.

Hence, our algorithm will correctly identify all the edges which belong to cycle, by checking if they are a bridge are not. The final output of the algorithm is then the least weight of all such edges.

2.6 Pseudo-Code

```
1: timer ← 0                                          ▷ Initialize global variable to store the timer
2: minWeight ← ∞                                      ▷ Initialize global variable to store the minimum required weight
3: procedure DFS(u, par)
4:     tin(u) ← timer
5:     low(u) ← timer
6:     timer ← timer + 1
7:     for v ∈ adj(u) do
8:         if v = par then                            ▷ Ignore the parent vertex
9:             continue
10:        else if tin(v) = −1 then                   ▷ Forward edge
11:            DFS(v, u)
12:            low(u) = min(low(u), low(v))
13:            if low(v) ≤ tin(u) then
14:                minWeight ← min(minWeight, min(getWeights(u, v)))
15:            end if
16:        else                                       ▷ Back edge
17:            low(u) = min(low(u), tin(v))
18:            minWeight ← min(minWeight, min(getWeights(u, v)))
19:        end if
20:    end for
21: end procedure
22: procedure getMinWeight
23:    for u ∈ V do
24:        if tin(u) = −1 then
25:            DFS(u, −1)
26:        end if
27:    end for
28:    if minWeight = ∞ then                          ▷ If there is no cycle in the graph, return −1
```

```
        minWeight ← −1
    end if
    return minWeight
end procedure
```

2.7 Time Complexity

Since we use a single DFS traversal of the graph, the Time Complexity of the solution is $O(|V| + |E|)$, where $|V|$ and $|E|$ are the number of vertices and edges respectively.

Since we know that $|V| = n$ and $|E| = n + 20$, the Time Complexity of the algorithm can be simplified to $O(n)$.

2.8 References

- Algorithm to find bridges

3 Question - 3

3.1 Problem

Suppose that G be a directed acyclic graph with following features.

- G has a s ingle source s and several sinks $t_1, ..., t_k$.
- Each edge $(v \to w)$ (i.e. an edge directed from v to w) has an associated weight $Pr(v \to w)$ between 0 and 1.
- For each non-sink vertex v, the total weight of all the edges leaving v is $\sum(v \to w) \in EPr(v \to w) = 1$
- The weights $Pr(v \to w)$ define a random walk in G from the source s to some sink t_i; after reaching any non-sink vertex v, the walk follows the edge $v \to w$ with probability $Pr(v \to w)$.
- All the probabilities are mutually independent.

Describe and analyze an algorithm to compute the probability that this random walk reaches sink t_i for every $i \in \{1, ..., k\}$. You can assume that an arithmetic operation takes $O(1)$ time.

3.2 Pre-Processing

We will represent the edges E as an array of arrays, with E_v being $I(v)$ ($u \in E_v$ if $(u \to v)$ is an edge in the graph). This is equivalent to rev_adj. We shall also initialize an array $memo$ of size $|V|$ to store all subproblems, and an answer array of size $|t|$ to store the final answers.

3.3 Approach

We shall use dynamic programmming, with the state $P(v)$ meaning the probability of a walk reaching vertex v, and the transitions given by $P(v) = \sum_{\exists i \to v} P(i) \cdot Pr(i \to v)$

3.4 Proof of Correctness

For any vertex v, let us denote $I(v)$ as the set of all vertices $x|(x \to v) \in E$. To reach the vertex v on a random walk, the path has to first reach a vertex $i \in I(v)$, then take the edge $(i \to v)$ to finally get to v. Therefore, the probability of reaching any vertex v $P(v)$ is
$P(v) = \sum_{i \in I(v)} P(i) \cdot Pr(i \to v)$
We can thus compute $P(t_i) \forall i \in 1, ..., k$ using dynamic programming

3.5 Conditions

- The sum of probabilities of outgoing edges from every node is 1, therefore, we can form a complete probability tree for every vertex.
- The source does not have any incoming edges
- Since this is a DAG, there aren't any circular dependencies in our recurrence relation

3.6 Subproblem

Our subproblem $P(v)$ is the probability of reaching the vertex v from s.

Therefore the subproblems for the final answers are $P(t_i) \forall i \in 1, ..., k$

3.7 Recurrence Relation

$P(v) = \sum_{i \in I(v)} P(i) \cdot Pr(i \to v)$

with our base case being $P(s) = 1$, since all walks start from there. For any individual path involving the edge $i \in I(v)$, the probability is $P(i) \cdot Pr(i \to v)$. We can just sum this over all $I(v)$ to get our final $P(v)$.

3.8 Graph Traversal

Since we solve every subproblem once, thus the corresponding vertex is picked and its immediate neighbours are visited next, this is equivalent to a DFS traversal of the graph.

3.9 Pseudo-Code

```
memo ← [−1, ..., −1]
procedure P(v,E)
    if memo[v] = −1 then
        s ← 0
        for i in E[v] do
            s ← s + P(i, E) · Pr(i, v)
        end for
        memo[v] ← s
    end if
    return memo[v]
end procedure
procedure SOLVE(E,s,t)
    memo[s] ← 1
    answer ← [−1, .., −1]
    for t_i in t do
        answer[i] ← P(t_i, E)
    end for
    return answer
end procedure
```

3.10 Time Complexity

Since there are V subproblems and to solve them we need to iterate through all edges once, the final time complexity is $O(V + E)$