Analysis and Design of Algorithms

Theory Assignment 2

Mehar Khurana (2021541) — Nalish Jain (2021543)

April 9, 2023

Question 1 — Part (a)

1.1 Description

The problem asks us to determine whether it is possible to legally drive from every intersection to every other intersection in a city with unidirectional streets. We can map the city as a directed graph G, with the intersections as vertices $v \in V$ and the streets as unidirectional edges $e \in E$. An example of the same has been shown below.

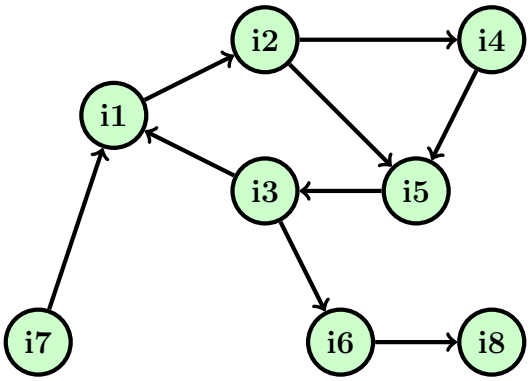


Figure 1: Nodes i1 - i8 represent intersections; arrows represent one-way streets

We hypothesize that the problem can be solved by checking if the entire graph G is a single Strongly Connected Component (SCC).

1.2 Algorithm

We can use Kosaraju's algorithm to find all the SCCs in the graph. If the graph has a single SCC, it means that we can legally drive from every intersection to every other intersection, i.e., there exists a path (of unidirectional edges) from every vertex in the graph to every other vertex. In case we encounter multiple SCCs, it means that there are some intersections that are not reachable from some other intersections, or some vertex pairs (v, u) do not have a path from v to u (or vice versa).

1.3 Why the algorithm works

Kosaraju's algorithm is inherently a linear time algorithm. In the first pass, we perform a DFS traversal of the graph to determine the finishing times of all vertices. As we do this traversal, we keep adding the vertices to the stack as we finish exploring them. This means that, in the case of a connected graph, the vertex we started at will be the one at the top. We then pop the topmost vertex from the stack and perform another DFS traversal of the reversed graph (all edge directions are reversed). As we visit the vertices in this traversal, we keep removing them from the stack.
Necessary and sufficient condition: If the stack is empty after we have finished one DFS traversal (i.e., all the other vertices are reachable from the topmost node, even on reversing the graph), we know that there is a single SCC in the graph, and the claim of the mayor is correct.

1.4 Pseudocode

Pseudocode pushed to next page due to less space.

1.5 Time Complexity

We perform two DFS passes in Kosaraju's algorithm, with each of them taking us $O(V + E)$ time to finish. Graph Reversal takes $O(E)$ time.

$$\implies \mathbf{T} = 2 * O(V + E) + O(E) = O(2V + 2E) + O(E) = O(V + E) + O(E)$$

$$\mathbf{T} = O(V + E)$$

Algorithm 1 Kosaraju's Algorithm for Finding Strongly Connected Components

```
1: function KOSARAJU(G)
2:     stack ← φ
3:     scc_count ← 0
4:     G_reverse ← REVERSE(G)
5:     for all v ∈ G.V do
6:         v.visited ← False
7:     for all v ∈ G.V do
8:         if not v.visited then
9:             DFS(G, v, stack)
10:
11:    for all v ∈ G_reverse.V do
12:        v.visited ← False
13:    while stack ≠ φ do
14:        v ← pop(stack)
15:        if not v.visited then
16:            scc_count ← scc_count + 1
17:            DFS_REVERSE(G_reverse, v)
18:    return (scc_count = 1) and (stack = φ)
19:
20: function DFS(G, v, stack)
21:    v.visited ← True
22:    for all neighbor ∈ G.E[v] do
23:        if not neighbor.visited then
24:            DFS(G, neighbor, stack)
25:    stack.push(v)
26: function DFS_REVERSE(G_reverse, v)
27:    v.visited ← True
28:    for all neighbor ∈ G_reverse.E[v] do
29:        if not neighbor.visited then
30:            DFS_REVERSE(G_reverse, neighbor)
31: function REVERSE(G)
32:    G'.V ← G.V
33:    G'.E ← φ
34:    for all (u, v) ∈ G.E do
35:        G'.E ← G'.E ∪ (v, u)
       return G'
```

Question 1 — Part (b)

1.1 Description

Now we are asked to determine whether this revised statement holds or not — *if you start driving from the 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.*

As we did in Part (a), we can map the city as a directed graph G, with the intersections as vertices $v \in V$ and the streets as unidirectional edges $e \in E$. Let the vertex denoting the town-hall be t. We can solve this problem by dividing all the other vertices (or intersections) into three groups. Any vertex $v \in V,\ v \neq t$ belongs to exactly one of the following groups:

1. vertex v is reachable from t, and t is reachable from v,
2. vertex v is reachable from t, but t is not reachable from v, and,
3. vertex v is not reachable from t.

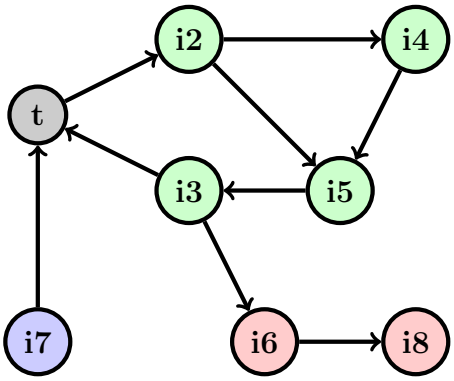
The mayor's statement does not specify any conditions for *group* 3, so it can be ignored, i.e., a 'favorable' graph may or may not have *group* 3 vertices (a 'favorable' graph is one that satisfies the mayor's new statement).

1.2 Algorithm and Explanation

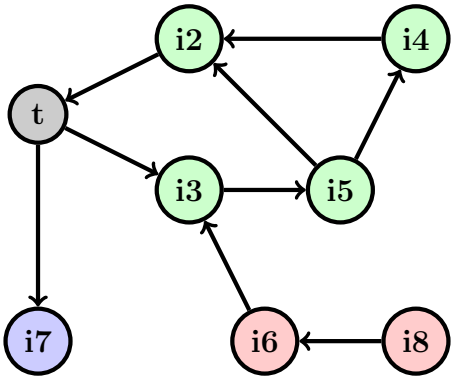
To find the number of all the vertices that belong to *group* 1, we run one pass of Kosaraju's algorithm (i.e., one forward DFS traversal, and one DFS traversal on the reverse graph, with both traversals originating from t). We maintain the count of all the vertices in the graph that belong to the same Strongly Connected Component as the vertex t. Note that there may be more than one SCC in a 'favorable' graph in part (b), however, we are only concerned with the SCC to which t belongs.

Now to count all the vertices that are reachable from t (i.e., vertices that belong to either *group* 1 or *group* 2), we run a DFS traversal once, which originates from t. This count can also be maintained in the forward pass of DFS used in Kosaraju's algorithm above.

We subtract the count attained from the first step from the count attained by the second step, to find the count of all the vertices in *group* 2. The problem statement clearly implies that there should not be any vertices in *group* 2 (necessary and sufficient condition). So, if the count of *group* 2 vertices equals 0, the mayor's new statement is correct.



(a) Forward Graph



(b) Reverse Graph

Figure 2: *group* 1 - green, *group* 2 - red, *group* 3 - blue

1.3 Pseudocode: Modified from Part (a)

Pseudocode pushed to next page due to less space.

1.4 Time Complexity

We perform two DFS (forward and reverse) in the above algorithm with each of them taking us $O(V + E)$ time to finish. Graph Reversal takes $O(E)$ time.

$$\implies \mathbf{T} = 2 * O(V + E) + O(E) = O(2V + 2E) + O(E) = O(V + E) + O(E)$$

$$\mathbf{T} = O(V + E)$$

Hence, the proposed algorithm is a linear-time algorithm.

Algorithm 2 Kosaraju's Algorithm (modified)

```
1: function KOSARAJU(G, t)
2:     stack ← φ
3:     forward_count ← 0          ▷ Equivalent to number of vertices reachable from t
4:     group_1_count ← 0          ▷ Equivalent to number of vertices in t's SCC
5:     G_reverse ← REVERSE(G)
6:     for all v ∈ G.V do         ▷ Forward DFS traversal from town-hall
7:         v.visited ← False
8:     DFS(G, t, forward_count, stack)
9:
10:    for all v ∈ G.V do         ▷ Reverse DFS traversal from town-hall
11:        v.visited ← False
12:    DFS_REVERSE(G, t)
13:    while stack ≠ φ do         ▷ Count vertices in the stack that have been visited
14:        v ← pop(stack)
15:        if v.visited then
16:            group_1_count ← group_1_count + 1
17:    return (forward_count − group_1_count = 0)
18:
19: function DFS(G, v, forward_count, stack)
20:    v.visited ← True
21:    forward_count ← forward_count + 1
22:    for all neighbor ∈ G.E[v] do
23:        if not neighbor.visited then
24:            DFS(G, neighbor, forward_count, stack)
25:    stack.push(v)
26: function DFS_REVERSE(G_reverse, v)
27:    v.visited ← True
28:    for all neighbor ∈ G_reverse.E[v] do
29:        if not neighbor.visited then
30:            DFS_REVERSE(G_reverse, neighbor)
31: function REVERSE(G)
32:    G′.V ← G.V
33:    G′.E ← φ
34:    for all (u, v) ∈ G.E do
35:        G′.E ← G′.E ∪ (v, u)   ▷ Add reverse edge to the reverse graph
       return G′
```

Question 2

2.1 Algorithm Description

1: **procedure** CycleEdgeMinWeight

2: First the algorithm stores the edges along with their corresponding weights in the set $edgeWeights$ in the form (u, v, w).

3: Then the algorithm uses the modified DFS traversal (Tarjan's algorithm [1]) to find all the bridge edges in the graph and store them in the set $Bridges$. The edges present in the set $Bridges$ are those edges which do not appear in any cycles, as on removing them, the graph gets broken into two components so they cannot appear in any cycle.

4: Then the algorithm removes all the edges present in set $Bridges$ from the set $edgeWeights$. Now the edges left in the set $edgeWeights$ are those edges which appear in any cycle of the graph.

5: The algorithm then iterates over the set $edgeWeights$ and returns the edge with the minimum weight along with its weight.

[1] Reference for Tarjan's Algorithm: https://youtu.be/qrAub5z8FeA

2.2 Tarjan's Algorithm

Algorithm 3 Tarjan's Algorithm

Inputs:
node: current node being explored
parent: parent node of the current node
visited: array to keep track of visited nodes
graph: adjacency list of the graph
time_in: array to store discovery time of each node
low: array to store the smallest time_in value reachable from each node

Outputs:
Bridges: a set containing all the bridges in the graph

```
1: timer ← 1
2: procedure TARJANSALGORITHM(node, parent, visited, graph, time_in, low, Bridges)
3:     visited[node] ← 1
4:     time_in[node] ← timer
5:     low[node] ← timer
6:     timer ← timer + 1
7:     for all i ∈ graph[node] do
8:         if i[0] = parent then
9:             continue
10:        if visited[i[0]] = 0 then
11:            weight ← i[1]
12:            TARJANSALGORITHM(i[0], node, visited, graph, time_in, low, Bridges)
13:            low[node] ← min(low[node], low[i[0]])
14:            if low[i[0]] > time_in[node] then
15:                Bridges.Insert(i[0], node, weight);
16:        else
17:            low[node] ← min(low[node], low[i[0]]);
```

2.3 Why the algorithm works

A bridge is an edge whose removal breaks the graph into two components, therefore an edge which occurs in a cycle can never be a bridge. Since after removing all the bridges in the graph using Tarjan's algorithm we are only left with edges present in the cycle. Edges with the minimum weight can be easily found in one iteration of the remaining edges. Hence the algorithm works correctly.

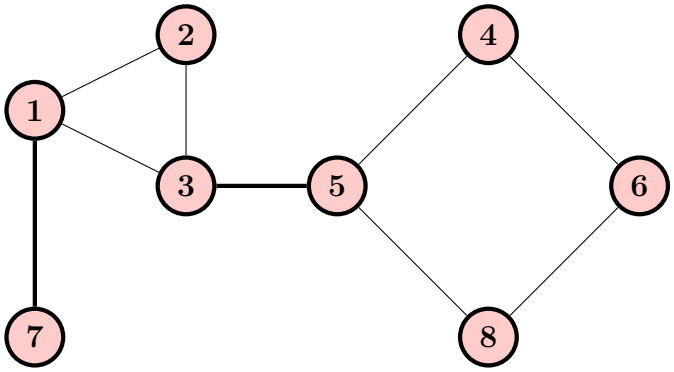


Figure 3: The darker edges are Bridges, and do not appear in any cycle

2.4 Time Complexity

Since the algorithm first finds the bridges in the graph using Tarjan's algorithm which is basically the modified version of DFS algorithm[O(V+E)]. Then it finds the edge with minimum weigth in O(E).

$$\mathbf{T} = O(V + E) + O(E)$$

$$\mathbf{T} = O(V + E)$$

Therfore overall time complexity is O(V+E).

Question 3

3.1 Algorithm Description

First, the algorithm sorts the vertices topologically by using the **DFS traversal algorithm** starting from the source S. The algorithm stores the topologically sorted vertices in an array *topoSort*. Then the algorithm creates a Probability array of $size(V)$ initialized with value 0 for all the vertices except source S which is initialized to 1. Then the algorithm starts from the source S (it will be the first vertex in the *topoSort* since its in-degree will be 0) and updates the probability of all the vertices connected to it by a direct edge by performing the following operation.
For example, in the case of edge (u, v) with $ProbabilityEdgeWeight\ w$,

$$Probability[v] = Probability[v] + w * Probability[u]$$

It iteratively performs this for all the vertices in the order given by topological sort. Finally the algorithm prints the value of all **sink** vertices by checking whether the outdegree of the vertex is 0 or not.
Note: The topological sort can be considered as a Preprocessing step

3.2 Input Format

Graph is stored in the form of adjacency list and adjacency list of a vertex "u" stores the adjacent vertex "v" and corresponding edge weight "w" in the form (v, w)

3.3 Pseudo Code

Algorithm 4 Topological Sort

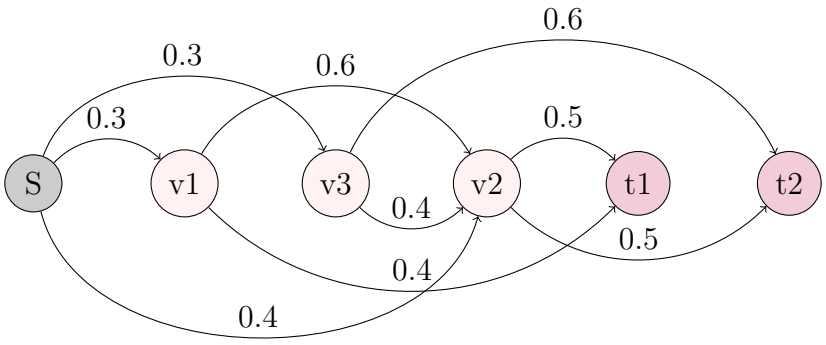
```
procedure TOPOLOGICALSORT(vertex, visited, graph, topoSort)
    visited[vertex] ← 1
    for each neighbor it of vertex in graph do
        if visited[it[0]] = 0 then
            topologicalSort(it[0], visited, graph, topoSort)
    push vertex into topoSort
procedure PROBABILITYCALCULATOR(Source, V, graph)
    visited ← array of V elements initialized to 0
    Probability ← array of V elements initialized to 0
    Probability[Source] ← 1
    topoSort ← empty stack
    topologicalSort(Source, visited, graph, topoSort)
    while topoSort is not empty do
        currentVertex ← pop top element from topoSort
        for i ← 0 to size of graph[currentVertex] − 1 do
            probabilityOfEdge ← graph[currentVertex][i][1]
            adjacentVertex ← graph[currentVertex][i][0]
            Probability[adjacentVertex] ← Probability[adjacentVertex]+
            Probability[currentVertex] ∗ probabilityOfEdge
    for i ← 0 to V − 1 do
        if size of graph[i] = 0 then
            print "Probability to reach sink " + i + " is " + Probability[i]
```

3.4 Why the algorithm works and Sub-problems

Once we start performing the operation in topological sort order, the final value of a vertex in the Probability array gives us the probability of reaching that vertex from source S.
Note: Since the vertices are topologically sorted a vertex cannot change the probabilities of the vertices behind it in the order as there is no outgoing edge from the current vertex towards them.
As going forward we multiply the edge weight (Probability of reaching the next vertex from the given vertex) with the probability of reaching the given vertex and add it to the probability of reaching the next vertex.
This can be considered as a **sub-problem** as we find the probability of reaching the next adjacent vertex from the probability of the given vertex using the recurrence relation.

$$\textbf{Recurrence Relation :}\ Probability[v] = Probability[v] + \sum_{(u,v)\in E} w * Probability[u]$$

Sub-problems which solve the **final problem** will be those in which vertex v is a sink vertex.
This can be better explained with the help of the given example.



3.5 Working of the Algorithm

Showing the values in Probability array after every iteration.

3.5.1 Iteration 0

Iterator at source S

1	0	0	0	0	0

3.5.2 Iteration 1

Source S updates the probabilities of the vertices v1, v2 and v3 which are directly adjacent to it.

1	0.3	0.3	0.4	0	0

3.5.3 Iteration 2

Iterator at v1 and it updates the probabilities of v2 and t1 directly adjacent to it.

1	0.3	0.3	0.58	0.12	0

3.5.4 Iteration 3

Iterator at v3 and it updates the probabilities of v2 and t2 directly adjacent to it.
The final probability of reaching v2 from source S is 0.4 + probability of reaching(v1 from source)*0.6 + probability of reaching(v3 from source)*0.4 as all the path that could have reached v2 from source S have been covered.

1	0.3	0.3	0.7	0.12	0.18

Similarly the algorithm updates the probabilities of reaching all the left vertices from source S.

3.5.5 Iteration 4

1	0.3	0.3	0.7	0.47	0.53

3.5.6 Iteration 5

1	0.3	0.3	0.7	0.47	0.53

3.5.7 Iteration 6

1	0.3	0.3	0.7	0.47	0.53

3.6 Time Complexity

The algorithim is doing **DFS traversal** once to topologically sort the vertices which takes time O(V+E).
Then the algorithim updates the Probability of vertices in time O(V+E) as the **For** loops runs for each edge once.
Then it finds the sink vertices and prints their probabilities in time V.
Therefore, the total time complexity is

$$\mathbf{T} = O(V + E) + O(V + E) + O(V)$$

$$\mathbf{T} = O(V + E)$$