Theory Assignment-2: ADA Winter-2023

Chaitanya Arora (2021033) Raghav Sakhuja (2021274)

---

1 Question

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.

(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.

1.1 Answer (a)

To mimic the scenario of the question, we can make a graph in which the roads are the directed edges in between the vertices, representing the roads' intersections. To solve part a, we can use Kosaraju's algorithm to find the SCCs in the graph.

1.1.1 Graph-Description

We change this question into a graph theoretic problem by converting the city into a graph with roads as directed edges between two intersections represented by the vertices in the graph. Here the task of finding if a person can drive from one intersection to any other intersection can be mimicked the task of finding if the graph is strongly connected or not i.e. if the graph has only one SCC.

1.1.2 Explanation of Algorithm

We use Kosaraju's algorithm to find the SCCs in the graph. On the first step we do a dfs.We start at each vertex of the graph and run a depth first search from every non-visited vertex until all the vertices are visited. For each vertex we are keeping track of exit time t_{out}. We save these exit times in an array and sort them in decreasing order.

Now, we create a new graph by going over every edge in the graph and reverse the direction of edges.

After this, we apply dfs, on the vertices according to their sorted order. The number of times we have to initiate a dfs gives us the number of strongly connected components of a graph. If this number is 1 (the initial dfs) , then the claim is true and we can move from any intersection to any other intersection.

1.1.3 Correctness

The definition of a strongly connected component is the maximumal set of vertices where there is a path from one vertex to any other vertex in that set. This is exactly what the claim of the mayor is. So if there are more than one SCC. Let the graph have k SCC's indexed from 1. Then if there is a path which connects SCC i with SCC j where $0 < i, j <= k$ then there cannot be a path from j^{th} SCC to i^{th} as that would make them a single SCC. i.e. it is not possible to have a path both ways, Therefore if there are more than 1 SCC, it would not be possible to go from any intersection to all other intersections.

1.1.4 Run-time

In this algorithm, we have to perform DFS twice, and iterate over all the edges once to reverse the directions.

Time complexity of DFS is:

$$O(E + V)$$

Time Comlexity of reversing directions:

$$O(E)$$

Thus the total Time Complexity is :

$$O(E + V) + O(E) + O(E + V)$$

$$O(E + V)$$

1.2 Answer (b)

To mimic the scenario of the question, we can make a graph in which the roads are the directed edges in between the vertices, representing the roads' intersections. To solve part b, we use dfs to find which all intersections are reachable and then reverse the directions of edges and redo the dfs to find the intersections that have a path to the townhall.

1.2.1 Graph-Description

We change this question into a graph theoretic problem by converting the city into a graph with roads as directed edges between two intersections represented by the vertices in the graph. Here the task to find whether there is a path from townhall to any other intersection and if there is a path, then it must also have a path back to the townhall.

1.2.2 Explanation of Algorithm

We make 2 arrays one checking if an intersection is Reachable and one checking if an intersection is Returnable. For Reachable, We simply run a dfs in the graph and mark every vertex that we can reach as reachable.

Next we iterate over all the edges and reverse them to create a new graph. In this all the roads would have opposite directions.

Now using this, we run a dfs again in the graph and mark every vertex that we can reach as returnable. These are the nodes that in the original graph have a path to the townhall, i.e one can return from that intersection to the townhall.

Now we iterate over the Reachable and Returnable arrays. If we find an instance where Reachable is true and Returnable is false, We return that the claim made the mayor is wrong. Else it means that there is no path from townhall to an intersection, or if it has one it will definitely have a path back.

1.2.3 Correctness

The Reachable array storing whether an intersection is reachable from townhall is trivial as it is the basic structure of Graphs and how dfs work. Now to show that in the reversed graph, if there is a path from townhall to the intersection means that there was a path from that particular intersection to the townhall in the reversed graph, we can use that fact if there is a path from node i to node j in a graph, there will be a path from j to i in the reversed graph. Thus using Reachable and Returnable arrays, we can tell if an intersection has a path from the townhall and to the townhall.

1.2.4 Run-time

In this algorithm, we have to perform DFS twice, and iterate over all the edges once to reverse the directions.

Time complexity of DFS is:

$$O(E+V)$$

Time Comlexity of reversing directions:

$$O(E)$$

Thus the total Time Complexity is :

$$O(E+V)+O(E)+O(E+V)$$

$$O(E+V)$$

2 Question

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.

2.1 Answer

We can modify the Articulation Point algorithm to make use of the concept of "time in" and "time out" during a depth-first search (DFS) traversal to solve the problem. We can identify whether an edge is part of a cycle in the Graph or not.

2.1.1 Pre-Processing State

Initialize time-in[ ] array of size equal to number of vertices and set all elements of time-in to INF.

Initialize low[ ] array of size equal to number of vertices and set all elements of low to INF

2.1.2 Explanation of the Algorithm

Perform a DFS travel on the graph and mark the vertex from which DFS travel is started as root of the DFS Tree. In the DFS Traversal mark the vertices as visited and keep a track of 'time-in'. For each vertex v visited during the DFS traversal, calculate the lowest "time in" reachable from v using a back edge (i.e., an edge that connects v to one of its ancestors in the DFS tree) or a forward edge. Let's denote this value as low[v].

This allows us find the back-egdes in the DFS tree. If a node u has a child v whose low[v] is more than than "time-in" of u, then it means that there is a back edge in the subtree of u. Meaning that (u,v) is definitely part of a cycle.

low[v] = min(tin[v],low[u])

An edge (u, v) is part of a cycle if and only if u and v are not connected by a parent-child relationship in the DFS tree, and u and v have the same low value, i.e., low[u] = low[v]. In the DFS recursion, we will update low values of vertices and also we can keep a track of min-weight and edge-with-min-weight.

2.1.3 Pseudo-code

Input: A edge-weighted connected undirected graph G = (V, E) with n + 20 edges.

Initialize time-in[] array of size equal to number of vertices and set all elements of $time_in$ to INF

Initialize low[] array of size equal to the number of vertices and set all elements of low to INF

Presumptions are as follows:-

1. edge(v,u) is the edge between v and u
2. weight(v,u) will return the weight of the edge(v,u)
3. min_{weight} = INF
4. $edge_{min}$=edge with min weight
5. The graph is of the form:

class Node
public int val;
public List (Node) neighbors;

Algorithm 1

```
1: function DFS(Vertex : u)
2:     mark u as visited
3:     time-in[u] = time
4:     low[u] = time
5:     time++
6:     for v in neighbors of u do
7:         if v is not visited then
8:             DFS(v)
9:         end if
10:        if low[v] >= low[u] then
11:            low[u] ← min(low(u), low(v))
12:            if weight(u, v) < min_weight then
13:                min_weight = weight(v, u)
14:                edge_min = edge(u, v)
15:            end if
16:        end if
17:    end for
18: end function
```

$edge_{min}$ will store the edge with the smallest weight min_{weight} contained in a cycle of G.

2.1.4 Justification

The algorithm correctly marks the edges of a cycle during the DFS traversal by comparing the "time in" and "low" values of the vertices connected by each edge. This is based on the fact that back edges in a DFS tree have the property that the "time in" value of the source vertex is greater than the "time in" value of the target vertex, and the vertices have the same low value. This property ensures that only edges that are actually part of a cycle are compared, and no false positives or false negatives are compared while calculating the smallest weight contained in a cycle of G.

The algorithm assumes that graph G is connected and undirected, with edge weights assigned. It also assumes that the graph has n + 20 edges, which implies that the graph has a sufficient number of edges to form cycles. The algorithm considers all the edges in the graph and correctly identifies the edge with the smallest weight contained in a cycle, if any.

2.1.5 Run-time

In this algorithm, we have to perform DFS twice, and iterate over all the edges once to reverse the directions.

Time complexity of DFS is:

$$O(E + V)$$

Here E = n + 20 and V = n,therefore:

Time complexity of algorithm would become:

$$O(n + n + 20)$$

The identification of cycle edges is done during the DFS traversal, and each step takes constant time, Thus its time complexity for all the steps in one iteration of the for loop inside the DFS would become:

$$O(1)$$

Thus the total Time Complexity is :

$$O(n + n + 20)$$

$$O(2 * n + 20)$$

$$O(n)$$

3 Question

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

• G has a single source s and several sinks t1, . . . , tk.

• Each edge $(v \to w)$ (i.e. an edge directed from v to w) has an associated weight $p(v \to w)$ between 0 and 1.

• For each non-sink vertex v, the total weight of all the edges leaving v is $\Sigma_{(v \to w) \epsilon E} Pr(v \to w) = 1$.

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 $i \epsilon 1, ..., k$. You can assume that an arithmetic operation takes O(1)-time.

3.1 Answer

3.1.1 Pre-Processing State

Make the graph G such that G is of the form of adjacency list. Here, we also reverse the directions of edges, so that instead of traversing from s to t_i, we can traverse directly from t_i to s.

Initialize an array dp of size n, where n is the number of vertices in the graph G, and initialize all elements with INF.

Set dp[s] = 1, where s is the source vertex of G

3.1.2 Sub-Problem

Define a recursive function Opt(vertex : u, graph : G) that takes a vertex u in G as input and returns the probability that the random walk starting from s reaches u.

3.1.3 Recurrence

Opt(u) = 1, if u = s // Base case: probability of reaching the source s from itself is always 1

Opt(u) = dp[u], if dp[u] != INF // If the probability is already computed, return it from dp[u]

Opt(u) = $\Sigma(weight(v, u) * Opt(v))$, for all v in adjacency list of u // Recurrence relation

The recurrence relation is defined in terms of Opt(u), which represents the probability of reaching vertex u from the source s. The base case is when u is the source s, in which case the probability is always 1. If the probability for vertex u is already computed and stored in dp[u], then it is directly returned. Otherwise, the probability is calculated using the sum of probabilities of reaching u from its adjacent vertices v, weighted by the edge weights. This recurrence relation forms the basis of the dynamic programming approach used in the given algorithm to compute the probabilities of reaching each vertex in the graph from the source s efficiently.

Algorithm 2 Walk of graph

Input: $G = (V, E)$: reversed graph, arr: the array containing all the sinks $t_1, t_2, ...t_k$, s the source vertex.
Initialize dp: array of size n(number of vertex) initialized with INF
Set $dp[s] = 1$

```
function OPT(vertex : u, graph : G)
    if dp[u]! = INF then
        return dp[u]
    end if
    sum ← 0
    for v in adjacency list of u do
        sum+ = weight(v, u) ∗ Opt(v)
    end for
    dp[u] ← sum
    return dp[u]
end function
```

3.1.4 Pseudo-code

3.1.5 Final Solution

The final solution to the problem can be get by running opt(t_i), where t-i is the ith sink and this would return the probability of reaching from s to sink.

3.1.6 Run-time

The time to reverse the graph is $O(E)$ we first reverse the graph in the pre-procession step. Then we use dynamic programming to go over every vertex. since we are saving the results of every vertex, we must go over every vertex only once. For every vertex, we go over all the edges that are connected to it. in worst case, this would mean all the edges are connected to the vertex. Thus the dp takes $O(E.V)$

therefore time complexity is

$$O(E) + O(E.V)$$

$$O(E.V)$$

Acknowledgement:
https://cp-algorithms.com/graph/strongly-connected-components.html for question 1.
https://cp-algorithms.com/graph/cutpoints.htmlalgorithm for question 2 (articulation points)