Theory Assignment-4: ADA Winter-2024

Aarzoo (2022008) Anushka Srivastava (2022086)

1 Algorithm Description

The algorithm to find the cut-vertices of a *Directed Acyclic Graph*. A vertex $v \notin \{s, t\}$ is called an (s, t)-cut vertex if every path from s to t passes through v. The algorithm to implement this is described below:

Step 1: The algorithm finds the topological sort of the vertices (i.e., arranging the vertices in the decreasing order of their post numbers) in the graph starting from the source vertex s till we reach the destination t.

The function `TopologicalSort(adj, sorted, visited, node, t)` takes the arguments as:

- `adj`: adjacency list
- `sorted`: returns the vertices in non-decreasing order of the post numbers
- `visited`: array which maintains the visited boolean of vertices
- `t`: destination vertex.

The sorted array is finally reversed to get the desired topological sort of the vertices. At the end, we have a list of vertices arranged in an order given by the topological sort algorithm.

Step 2: To find whether a given vertex v is a cut vertex, we need to check that there exists no edge $U \to W$ where U is strictly earlier than v and W is strictly later than v in the topological ordering.

To check this, we find the maximum vertex a particular vertex can reach by referring to its adjacency list. If all vertices before v can reach a maximum vertex of v itself, this indicates that all paths passing between s and t **have** to pass through v, which is a cut vertex. We have implemented the steps described below to find all the cut vertices using the above-described algorithm.

We first map all the vertices to their corresponding positions in the topological order in the function `FindCutVertices(adj, n, m, src, t)`. This helps us to track the position of the starting and ending vertex and the maximum index each vertex can reach. A variable `maxDist` maintains this maximum index. We iterate through all the vertices between s and t, and for each vertex, we update the `maxDist` to the maximum possible index, ensuring that it does not exceed the index of t. This step ensures that we only check the cut vertices for the vertices that lie in the path between s and t. If we reach a certain vertex v such that the maximum possible vertex for all the vertices before v is v itself, we add it to our list of cut vertices. These steps are then repeated further until we reach the final vertex t.

At the end of all these steps, we have a list that contains all of our cut vertices, through which all paths between s and t pass.

2 Complexity Analysis

2.1 Time Complexity

The time complexity of the algorithm is $O(n + m)$, i.e., $O(V + E)$.

The algorithm is divided into two steps:

Step 1: Finding the topological sort of the vertices is a modified implementation of *Depth First Search* algorithm, which takes $O(V + E)$ time.

Step 2: Iterating and storing the indices of all the vertices in the *index* map takes $O(V)$ time. To find the cut vertices, we iterate through our topologically sorted list and to find the maximum vertex any vertex can reach, we iterate through its adjacency list. So, overall, this step is equivalent to iterating through the entire adjacency list of the graph. Hence, in the worst case scenario, the time taken for this step would be $O(V + E)$. Hence, the overall time complexity of the entire algorithm is $O(V + E)$.

2.2 Space Complexity

The space complexity of the algorithm is $O(n + m)$, i.e., $O(V + E)$.

- `adj`: adjacency list occupies $O(V + E)$ space.
- `visited`: An array which maintains the boolean to check if the vertex is visited or not occupies $O(V)$ space.
- `sorted`: An array which contains the topological sort of the vertices, occupies $O(V)$ space.
- `index`: map which contains the mapping of the vertices to their corresponding index in the topological order, occupies $O(V)$ space.

Thus, overall space complexity is $O(V + E)$, i.e., $O(n + m)$.

3 Pseudocode

Illustrated below

Algorithm 1 Algorithm to find cut vertices between two given points

```
1: function TopologicalSort(adj, sorted, visited, node, t)
2:     visited[node] ← true
3:     if node = t then
4:         sorted.push_back(t)
5:         return
6:     end if
7:     for neighbour in adj[node] do
8:         if not visited[neighbour] then
9:             TopologicalSort(adj, sorted, visited, neighbour, t)
10:        end if
11:    end for
12:    sorted.push_back(node)
13: end function

14: function FindCutVertices(adj, n, m, src, t)
15:    sorted ← empty vector
16:    visited ← vector of (n + 1) elements initialized to false
17:    TopologicalSort(adj, sorted, visited, src, t)
18:    reverse(sorted)
19:    index ← empty map
20:    for i ← 0 to sorted.size() do
21:        index[sorted[i]] ← i
22:    end for
23:    maxDist ← 0
24:    cutVertex ← empty vector
25:    for i ← 1 to sorted.size() − 1 do
26:        if i > index[t] then
27:            break
28:        end if
29:        for j in adj[sorted[i - 1]] do
30:            if index[j] > maxDist and index[j] < index[t] then
31:                maxDist ← index[j]
32:            end if
33:        end for
34:        if maxDist ≤ i and maxDist ≠ 0 and sorted[i] ≠ src and sorted[i] ≠ t then
35:            cutVertex.push_back(sorted[i])
36:        end if
37:    end for
38:    return cutVertex
39: end function
```