Theory Assignment-4: ADA Winter-2024

Shobhit Raj (2022482) Vashu (2022606)

1 Algorithm Description

1.1 Pseudocode

Algorithm 1 Cut Vertices of a DAG (Continued)

Input: Graph G, $Adj[]$ list of G, n (no. of vertices), m (no. of edges), s (source vertex), t (destination vertex)
Output: The Cut Vertices of G
Assumption: 1-based indexing in all arrays and The vertices/nodes of the graph are given as integers i.e. $1, 2, 3, \ldots, n$ i.e. s and t are some integers
Initialization:

```
if n == 2 then
    return No Cut vertices present
end if
Initialize arrays of size n (i.e., no. of vertices) namely Pre, Post, Color, Parent, Toposort, Store, Location-Track, Cuts
Pre ← []
Post ← []
Color ← []
Parent ← []
Toposort ← []
Store ← []
LocationTrack ← []
Cuts ← []
Initialize an empty Stack named DFSFinished
```

Code:

```
function DFS(G)
    for all u ∈ V(G) do
        Color[u] ← red
    end for
    Time ← 0
    for all u ∈ V(G) do
        if Color[u] == red then
            DFSVisit(G, u)
        end if
    end for
end function

function DFSVisit(G, u)
    Color[u] ← blue                          ▷ vertex u is visited
    Time ← Time + 1
    Pre[u] ← Time
    for all v ∈ Adj[u] do
        if Color[v] == red then
            Parent[v] ← u
            DFSVisit(G, v)
        end if
    end for
    Color[u] ← green                         ▷ vertex u is explored completely
    Stack.push(u)
    Time ← Time + 1
    Post[u] ← Time
end function
```

//Assumption - We will run here DFSVisit(G,s) here once, if T is not visited, that means no path exists between s to t. Hence, we return 0 here. We have explained wwhy this happens in the explanation section. - This DFS takes O(V+E), so it doesn't affect the time complexity.

Algorithm 2 Cut Vertices of a DAG (Continued)

```
while !DFSFinished.empty do
    Node ← DFSFinished.top()
    Toposort.append(Node)
    DFSFinished.pop()
end while
// Now, we have the topological sort ordering of the DAG in the Toposort array (obtained using decreasing
sequence of postnumbers in the DFS traversal as taught in lecture)

for i = 1 to n do
    locationTrack[Toposort[i]] = i;
end for
i← 0
for all u ∈ Toposort do
    maxIndexLocation = −∞;
    for all v ∈ Adj[u] do
        maxIndexLocation = max(MaxIndexLocation, LocationTrack[v]);
    end for
    if i ≥ 1 then
        Store[i] = max(MaxIndexLocation, Store[i − 1]);
    else
        Store[i] = MaxIndexLocation;
    end if
    i ← i + 1
end for
CutCounter← 0
for i = |location_track[s]| to |location_track[t]| do
    if store[i − 1] <= i then
        CutCounter ← CutCounter + 1
        Cuts.append(Toposort[i])
    end if
end for
return Cuts
```

1.2 Detailed Explanation of the Algorithm

First, observe that every vertex v can reach t: We can repeatedly follow outgoing edges starting at v, and t will be the only vertex where there won't be an outgoing edge to follow. Similarly, s can reach every vertex v. If there are only 2 vertices present s and t, then there will be no cut vertices in this graph. Also, if there is no path from s to t in the DAG, then there are no vertices of the graph which will be in the path from s to t as no such path exists, hence the condition of cut vertex is violated, hence to cut vertices in the graph. Now, consider a topological ordering $<$ of G's vertices. Per the advice, we claim v is an (s, t)-cut vertex if and only if there exists no edge $u \to v$ where $u < v$ and $v < w$. Suppose there does exist such an edge $u \to w$. Then, there is a bridge B from s to u, along $u \to$, and from w to t. Bridge B does not include v, because its vertices must appear in topological order and v is neither before u nor after w in ordering $<$. Therefore, v is not an (s, t)-cut vertex. Now, suppose there is no such edge. Every path from s to t follows its vertices in topological order. We cannot "skip" v along any path because otherwise there would be an edge going directly from an earlier vertex to a later vertex in the ordering. We conclude that v is an (s, t)-cut vertex in this case.

We now use the following algorithm based on the observation. We will scan the vertices and will construct a *store[]* array which stores the information about the location of the farthest located node connected with outgoing edges in the *Toposort[]* array. Then, we check for each index i in whether there exists a node in the right of that index.

Some common Notation we used:

- **What is** $location\_track[]$**?**
  It's a mapping which is a correlation between each vertex of graph G and its corresponding index in the topological sorting arrangement of vertices. For example, $location\_track[t]$ gives me the location of node/vertex t in *Toposort[]* array.

- **What is** $store[i]$**?**
  It stores the location of the rightmost neighbor vertices "till" index i.

- **Then, what do you mean by rightmost neighbor?**
  Rightmost neighbor of a vertex v in graph G means the "farthest" index location for each u in the adjacency list of v in *Toposort[]* array.

After constructing *Toposort[]* array using DFS, we will construct *store[]* array in $O(|V| + |E|)$. We traverse for every element u in *Toposort[]* array, mark max_index_loc as $-\infty$, then we traverse through its every vertex connected to its outgoing edges (i.e., $adj[u]$) and will find which vertex has the farthest located. In the end, we compare it with the $store[i-1]$, therefore storing the index of farthest located vertices till index i.

In the end, we run a for loop from $|location\_track[s]|$ to $|location\_track[t]|$, that is, we are traversing only through the paths from s to t. Then we check for the condition of cut vertices by comparing the location of farthest located nodes connected to all the vertices to the left of that index using $store[i-1]$, which we have already constructed in $O(1)$ time.

2 Run time analysis

First, we are computing a topological order of the DAG using the DFS traversal algorithm which takes O(V + E) time. Then, forming the Toposort and locationTrack array takes O(V) time. Then, for each vertex u in the topological ordering, we are finding the maximum index of an outgoing edge from u, which requires traversing all the edges of the vertex u, this process in worst case take O(V + E) time. Finally, we are getting the cut vertices by running loop over the vertices in topological order from s to t, if the maxIndex of the just the left neighbour of the vertex is more than the index of the vertex, than a bridge edge exists. Else, the vertex is cut and is appended into the array. This comparison of index takes O(1) time, so the total loop takes O(V) time.

Overall, the runtime complexity of the provided approach is $O((V + E))$.