Theory Assignment-4: ADA Winter-2024

Vikranth Udandarao (2022570) Ansh Varshney (2022083)

1 Preprocessing and Assumptions

In this problem, we preprocess, assume or have been given the following:

1. The graph $G = (V, E)$ is a directed acyclic graph (DAG) and is given to us in the problem.
2. Two vertices s and t are specified in the graph.
3. The graph has n vertices and m edges.
4. The vertices of the graph are labeled from 0 to n-1.
5. A vertex v is defined as (s,t) cut if it is present in all the paths that can be possible from s to t. Vertex v cannot be s or t.
6. Graph has been given to us in the form of an adjacency list, i.e. stores the list of outgoing vertices for each vertex.
7. In the Sample Test Case, Vertex a is s (*Source*) and Vertex e is t (*Destination*).

2 Problem Formulation as a Graph Theoretic Problem

We know that we must find the vertices in all paths from s to t. This is equivalent to saying that on the removal of any of these vertices, we cannot reach from s to t. So, we formulate the problem as finding all vertices that, when removed, disconnect the path from s to t in a directed acyclic graph (DAG). This can be represented by constructing a graph where vertices are the nodes of the original DAG and edges represent the dependencies between nodes. If we take a theoretical example, nodes represent tasks in a project scheduling scenario, and directed edges represent dependencies between tasks. We can construct the graph from the adjacency list that we assume will be given to us. It has the outgoing vertices for each vertex of the graph.

Example: Consider a graph G with vertices s, A, B, C, and t such that $s \to A \to B \to C \to t$. Here, vertices A, B, and C are (s, t)-cut vertices as removing any of them disconnects the path from s to t.

3 Algorithm Description

We traversed the graph twice for this algorithm, each time using DFS. In the first traversal, we found a topological sorting of all the vertices (using DFS and stack). Since graph G is DAG, it can be linearly ordered topologically, and its topological sort will contain all the vertices. Then, we used a modified DFS based function FindCutNodes to identify which vertices are (s,t) cut.

3.1 Topological Sort Description

The Topological Sorting has been done in the following way:

1. Initialize an empty stack that will store the topological sorting.
2. Initialize a vistedtopo array that has n elements all initialized as 0 initially. This array will have value 0 if a vertex is not visited and 1 if it has been visited.
3. Perform a DFS traversal starting from vertex s.

4. When visiting a vertex v during DFS, mark it as visited,i.e., visitedtopo[v]=1.
5. Now iterate through all its neighboring vertices and call the DFS function recursively for all the vertices u that are not visited, i.e., visitedtopo[u]=0.
6. After iterating through all the neighbors of v, push v into the stack.
7. At the end, after all the vertices have been visited, the stack (from top to bottom following LIFO) will contain the topological sorted order of all the vertices.

3.2 Modified DFS (FindCutNodes) Description

The FindCutNodes is as follows:

1. Initialize an empty set or list to store cut vertices. Call it cutnodes.
2. Initialize an array start with n elements all equal to 0 initially. This array will store 1 if a vertex is visited (basically denotes entry during DFS).
3. Initialize an array end with n elements all equal to 0 initially. This array will store 1 if a vertex has been visited back, i.e., recursive calls for all the neighbors of this vertex have been made, and we have returned back to this vertex (basically denotes exit during DFS).
4. Perform DFS traversal in the topologically sorted order,i.e., call the FindCutNodes function for stack elements (in LIFO order) whose start value is 0 and pop the element from stack after the function has been called.
5. When visiting a vertex v during FindCutNodes function:
    - Update start[v]=1.
    - For each outgoing edge (v, u):
        - If u has start[u] equal to 0 (the node has not been visited before) call the function again recursively for this node.
        - Else if suppose start[u] is 1 but end[u] is 0, that means that our necessary condition for (s,t) cut vertex has been satisfied. This implies that u is a (s,t) cut vertex. Add it to the cutnodes array.
6. After this, mark end[v]=1, showing that the traversal for this vertex has been done.
7. After calling this function for each vertex, the cutnodes array will contain all the nodes that when removed will lead to no path left between s and t in the graph, i.e. all the (s,t) cut vertices.

4 Pseudocode

5 Complexity Analysis and Explanation of Running Time

5.1 Time Complexity

The time complexity of the algorithm is $O(n + m)$, where n is the number of vertices and m is the number of edges in the graph. This complexity arises from the following key operations:

- Topological sorting using DFS: This operation traverses each vertex and its outgoing edges once, leading to a time complexity of $O(n + m)$.
- DFS traversal in the findCutNodes function: This operation also traverses each vertex and its outgoing edges once, contributing to the overall time complexity of $O(n + m)$.

Algorithm 1 findCutNodes

```
function TopologicalSort($node, st, visited, adj[]$)
    $visited[node] \leftarrow 1$                                   ▷ Mark the current node as visited
    for $i$ in $adj[node]$ do                                     ▷ Iterate through the adjacent nodes
        if $visited[i] = 0$ then                                  ▷ If the adjacent node is not visited
            TopologicalSort($i, st, visited, adj[]$)              ▷ Recursively call TopologicalSort
        end if
    end for
    $st.push(node)$                                               ▷ Push the current node into the stack
end function

function findCutNodes($node, start, end, cut\_nodes, adj[]$)
    $start[node] \leftarrow 1$                                    ▷ Mark the current node as visited (entry)
    for $i$ in $adj[node]$ do                                     ▷ Iterate through the adjacent nodes
        if $start[i] = 0$ then                                    ▷ If the adjacent node is not visited (entry)
            findCutNodes($i, start, end, cut\_nodes, adj[]$)      ▷ Recursively call findCutNodes
        else if $start[i] = 1$ and $end[i] = 0$ then              ▷ If the adjacent node is visited but not exited
            $cut\_nodes.push\_back(i)$                            ▷ Add the current node to the cut nodes list
        end if
    end for
    $end[node] \leftarrow 1$                                      ▷ Mark the current node as exited
end function

function Main
    $visited\_topo \leftarrow$ vector$(n, 0)$                     ▷ Initialize visited array for topological sort
    $adj[] \leftarrow$ vector of adjacency lists                  ▷ Initialize adjacency lists
    $st \leftarrow$ empty stack                                   ▷ Initialize empty stack for topological sort
    for $i$ from 0 to $n - 1$ do                                  ▷ Iterate through all nodes
        if $visited\_topo[i] = 0$ then                            ▷ If the node is not visited
            TopologicalSort($i, st, visited\_topo, adj[]$)        ▷ Call TopologicalSort for unvisited nodes
        end if
    end for
    $start \leftarrow$ vector$(n, 0)$                             ▷ Initialize start array for findCutNodes
    $end \leftarrow$ vector$(n, 0)$                               ▷ Initialize end array for findCutNodes
    $cut\_nodes \leftarrow$ empty vector                          ▷ Initialize empty vector for cut nodes
    while $st$ is not empty do                                    ▷ While stack is not empty
        $node \leftarrow st.top()$                                ▷ Get the top node from the stack
        $st.pop()$                                                ▷ Remove the top node from the stack
        if $start[node] = 0$ then                                 ▷ If the node is not visited (entry)
            findCutNodes($node, start, end, cut\_nodes, adj[]$)   ▷ Call findCutNodes for unvisited nodes
        end if
    end while
    return 0                                                      ▷ Return success
end function
```

5.2 Space Complexity

The space complexity of the algorithm is $O(n)$, primarily due to the following data structures and variables:

- Visited array (visited_topo): This array tracks visited nodes during topological sorting and DFS traversal, requiring $O(n)$ space.
- Start and end arrays in findCutNodes: These arrays store information about node traversal states, requiring $O(n)$ space each.
- Stack (st): The stack is used for topological sorting and can store up to n vertices, adding $O(n)$ space complexity.

6 Necessary and Sufficient Conditions explaining the algorithm

Any vertex v that is (s,t)-cut vertex will have all paths from s to t pass through v. So, if suppose v is removed from the graph, then there will be no path to reach from s to t. Hence, we need to find the vertices that, when removed from the graph will make t unreachable from s in the graph.

We have first ordered the DAG in a topologically sorted order so that we do not randomly iterate through the vertices later, as this might increase the time complexity. Topological sorting ensures that every vertex comes before its successors in the ordering. This is crucial because it helps efficiently identify each vertex's ancestors and descendants.

Then, for a vertex v to be cut vertex, it should not have a back edge from any of its descendants to any of its ancestors. If this is true, then there should not be any child u of v that has a back edge from u's descendant to v's ancestor. This ensures that removing v would disconnect the path between s and t. If there were a back edge from a descendant of v to an ancestor of v, it would create an alternative path bypassing v. Hence, v wouldn't be a cut vertex. We check these conditions by checking that the vertex's start array value is 1 and the end array value is 0. If this is true, then the vertex will be a cut vertex and we will add it to the cutnodes array.

7 Proof of Correctness

Proof of Correctness by Induction:

We will prove the correctness of the algorithm using induction.

Base Case (k = 1)

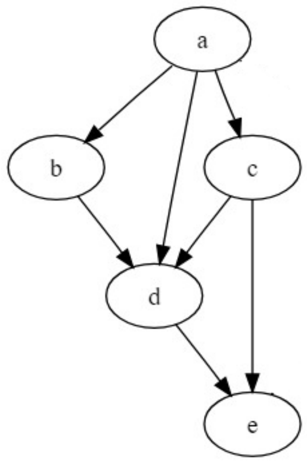
For a graph with only one vertex, the algorithm correctly identifies that the vertex is not an (s, t)-cut vertex because it cannot disconnect any path between s and t (since there is no other vertex to connect to).

Inductive Step

Assume that the algorithm correctly identifies (s, t)-cut vertices for a graph with k vertices. We will now prove that it also works for a graph with $k + 1$ vertices.

Let's denote the graph with k vertices as G_k and the graph with $k + 1$ vertices as G_{k+1}. Now, we add one more vertex to G_k to form G_{k+1}.

- **Case 1:** The new vertex added is not connected to any existing vertex.
  - In this case, the new vertex cannot be an (s, t)-cut vertex because it does not lie on any path from s to t. The algorithm correctly identifies this and does not include it in the cut nodes list.
- **Case 2:** The new vertex is connected to existing vertices in G_k.
  - Now, let's consider the scenarios:
    - If the new vertex is connected to s or t:
      - In this case, the new vertex cannot be an (s, t)-cut vertex because removing it does not disconnect any path from s to t.
    - If the new vertex is connected to vertices other than s and t:



· In this case, removing the new vertex may or may not disconnect paths from s to t, depending on the structure of the graph. The algorithm correctly identifies this by performing DFS traversal and finding all (s, t)-cut vertices.

By the induction hypothesis, the algorithm correctly identifies (s, t)-cut vertices for G_k. And from the above analysis, we see that it also correctly handles the addition of one more vertex to form G_{k+1}. Therefore, by induction, the algorithm correctly identifies (s, t)-cut vertices for any graph G with $k + 1$ vertices.

This completes the proof of correctness by induction.

Soundness: The algorithm is sound because it only identifies a vertex as an (s, t)-cut vertex if the removal of this vertex increases the number of disconnected components, meaning there is no path from s to t that bypasses this vertex. This is verified by the DFS traversal which ensures that each identified cut vertex is on all paths from s to t.

Completeness: The algorithm is complete because it performs a thorough DFS traversal starting from the source s, considering all possible paths to the destination t. During this traversal, it checks every vertex that could potentially be an (s, t)-cut vertex by exploring all outgoing edges. Thus, it guarantees that no potential cut vertex is missed.

CHECK BELOW FOR SAMPLE TESTCASE

SAMPLE TESTCASE:

Vertices:

- a
- b
- c
- d
- e

Edges:

- a $\rightarrow$ b
- a $\rightarrow$ d
- a $\rightarrow$ c
- b $\rightarrow$ d
- c $\rightarrow$ d
- c $\rightarrow$ e
- d $\rightarrow$ e

Expected Output:
Cut nodes: d
Explanation:

- Vertex a is the source, and vertex e is the destination.
- Vertex d is a cut vertex because it is the only point of failure for the paths $a \to d \to e$, $b \to d \to e$, and $c \to d \to e$. Without d, there is no path from a, b, or c to e, thus increasing the number of disconnected components from the source to the destination.
- Vertices b and c are not cut vertices with respect to the source a and destination e. While they do participate in directing the flow of the graph, their removal does not isolate e from a due to alternative paths available ($a \to d \to e$ in case of removing b, and $a \to b \to d \to e$ in case of removing c).

Therefore, the removal of vertex d would prevent access from the source a to the destination e, confirming it as the critical cut vertex in the context of directed paths between a and e.