Theory Assignment-4: ADA Winter-2024

Shamik Sinha (2022468) Vansh Yadav (2022559)

Assumptions Made

Our algorithm for finding (s, t)-cut vertices in a directed acyclic graph (DAG) assumes the following:

1. The graph is represented by an adjacency list where each vertex v_i is associated with a list of vertices it is adjacent to, denoted by $adj[v_i]$.
2. The graph consists of n vertices, numbered from 0 to $n - 1$.
3. The vertices s and t are part of the graph, and they are distinct from each other and from all other vertices in the graph.
4. There is always a path between vertices s and t. (If there is no path from s to t in the directed acyclic graph G, then there cannot be any (s, t)-cut vertices. This is because a cut vertex is defined as a vertex that lies on every path from s to t. If there are no paths from s to t, then there cannot be any vertices that lie on every such path.)
5. Each edge in the graph is directed, and there are no self-loops.
6. The graph is acyclic, meaning there are no cycles present.

Input

The input describes a directed acyclic graph (DAG) with the following properties:

- **Number of Vertices** (n)
- **Number of Edges** (m)
- **Source Vertex** (s)
- **Target Vertex** (t)

The edges of the graph are provided in the following format:

$$[\text{vertex\_from}] \rightarrow [\text{vertex\_to}]$$

Each entry represents a directed edge from a source vertex to a target vertex.

Output

The output consists of a list of vertices that are (s, t)-cut vertices in the directed acyclic graph (DAG). The format is as follows:

- The output is a single line containing space-separated integers representing the (s, t)-cut vertices.
- Each integer corresponds to a vertex that lies on all paths from the source vertex s to the target vertex t, excluding s and t themselves.

Algorithm Description

The algorithm is designed to find all the vertices in a directed acyclic graph (DAG) that act as (s, t)-cut vertices, i.e., vertices that must be traversed in order to go from the source vertex s to the target vertex t. Below are the main steps of the algorithm:

1. **Input Reading**:
   - Read the number of vertices n and the number of edges m from the input.
   - Read the source vertex s and the target vertex t from the input.
   - Construct the adjacency list representing the graph.
2. **Main Loop**:
   - Iterate over all vertices except for s and t, since they cannot be cut vertices i.e. are always included in the path from s to t.
   - Create a vector `visited` to mark visited nodes during the DFS traversal.
   - Call the `isCutVertex` function for each vertex to determine if it is a cut vertex.
   - If a vertex is not found to be necessary to reach t, add it to the list of cut vertices (`cutVertices`).
3. **DFS Function** `isCutVertex`:
   - This function performs a Depth-First Search (DFS) traversal starting from the source vertex s to determine if a given vertex i is a cut vertex.
   - It traverses the graph recursively, marking visited nodes and avoiding the deleted node (passed as an argument).
   - If the destination vertex t is reached during traversal, it returns `true`, indicating that the vertex is necessary to reach t. Therefore, any path between s and t must pass through this vertex i.e. it is a cut vertex.
   - Otherwise, it returns `false`, indicating that the vertex is not necessary to reach t and therefore not a cut vertex.
   - The recurrence in the DFS function `isCutVertex` arises from its recursive nature:
     (a) **Base Case**: When the function reaches the destination vertex t, it returns `true`, indicating that the vertex is necessary to reach t, and thus, it is a cut vertex.
     (b) **Recursive Case**: If the function has not reached t, it continues the DFS traversal by exploring neighboring vertices recursively.

     The recurrence occurs as the function explores neighboring vertices until it either reaches t or exhausts all reachable vertices.
4. **Output**:
   - Print the list of cut vertices (`cutVertices`).

The main idea behind the algorithm is to perform a DFS traversal from s to t, excluding one vertex at a time and checking if t is still reachable. If t is reachable after excluding a vertex, it implies that the excluded vertex is necessary to traverse from s to t, making it a cut vertex. The algorithm employs a depth-first search approach to efficiently explore the graph and identify the cut vertices.

Time and Space Complexity

Time Complexity Analysis

The running time of the provided algorithm can be analyzed as follows:

1. **Input Reading**: Reading the number of vertices, number of edges, source vertex, and target vertex takes constant time, denoted as $O(1)$.
2. **Constructing Adjacency List**: Constructing the adjacency list from the input edges requires iterating over all edges once, which takes $O(m)$ time, where m is the number of edges.
3. **Main Loop**:
   - The main loop iterates over all vertices except s and t, which takes $O(n)$ time.
   - Inside the loop, a DFS traversal is performed using the `isCutVertex` function, which takes $O(n+m)$ time per iteration.
   - Therefore, the total time complexity of the main loop is $O(n^2 + mn)$.
4. **DFS Function** `isCutVertex`:
   - The DFS function traverses the graph recursively from the source vertex to determine if a given vertex is a cut vertex. In the worst case, each vertex and each edge may be visited once, resulting in $O(n+m)$ time complexity, where n is the number of vertices and m is the number of edges.

     The worst case occurs when the algorithm has to traverse through all the nodes in the graph. Therefore the sum of the vertices(n) and the edges(m) is the worst-case scenario. This can be expressed as O(n + m).
   - This function is called for each vertex except s and t, leading to a total worst-case complexity of $O(n * (n + m))$.
5. **Output**: Printing the list of cut vertices takes linear time with respect to the number of cut vertices, which is $O(n)$ in the worst case.

Combining all the steps, the overall time complexity of the algorithm is dominated by the DFS traversal within the main loop. Therefore, the explicit polynomial running time of the algorithm is $O(n^2 + mn)$, where n is the number of vertices and m is the number of edges in the graph.

This running time is consistent with the algorithm's design, as it involves traversing the graph using DFS, which has a time complexity proportional to the number of vertices and edges visited. Since the graph is a directed acyclic graph (DAG), the DFS traversal ensures that each vertex and edge is visited at most once, resulting in a polynomial time complexity.

Space Complexity Analysis

The space complexity of the algorithm:

1. **Adjacency List**: The adjacency list requires $O(n + m)$ space to store the graph, where n is the number of vertices and m is the number of edges.
2. **Visited Array**: Within the `isCutVertex` function, a visited array of size n is created to mark vertices as visited during the DFS traversal. Therefore, it requires $O(n)$ space.
3. **Main Loop Variables**: Additional variables used within the main loop, such as loop counters and the list of cut vertices, require negligible space compared to the size of the graph.
4. **DFS Stack**: The DFS function utilizes recursion, which results in a call stack. The maximum depth of the call stack corresponds to the depth of the DFS traversal, which can be at most n in the worst case (when the entire graph needs to be traversed). Therefore, the space complexity contributed by the call stack is $O(n)$.

Combining all the components, the total space complexity of the algorithm is dominated by the space required for the adjacency list and the visited array, resulting in $O(n + m)$ space complexity.

This space complexity analysis indicates that the space required by the algorithm grows linearly with the size of the input graph, making it efficient in terms of memory usage.

Pseudocode

Below is the pseudocode which represents the logic of the C++ code we had written for the problem, in a more abstract and readable form.

Algorithm 1 Algorithm to Determine Cut Vertices between two vertices s and t

```
function isCutVertex($adj[], visited[], node, deleted, destination$)
    $visited[node] \leftarrow 1$
    for $it$ in $adj[node]$ do
        if $it == destination$ then
            return true
        end if
        if $it == deleted$ then
            continue                                        ▷ Skip if the node is deleted
        end if
        if not $visited[it]$ then
            if isCutVertex($adj, visited, it, deleted, destination$) then
                return true
            end if
        end if
    end for
    return false
end function

function main
    $n, m \leftarrow$ input("Enter number of vertices: "), input("Enter number of edges: ")
    $s, t \leftarrow$ input("Enter source and destination vertices: ")
    $adj[] \leftarrow$ array($n + 2$)                              ▷ Adjacency list
    for $i \leftarrow 0$ to $m - 1$ do
        $u, v \leftarrow$ input
        append($adj[u], v$)                                 ▷ Adding edge to adjacency list
    end for
    $cutVertices \leftarrow$ array                                ▷ Vector to store cut vertices
    for $i \leftarrow 0$ to $n - 1$ do
        if $i \neq s$ and $i \neq t$ then
            $visited[] \leftarrow$ array($n, 0$)
            if not isCutVertex($adj, visited, s, i, t$) then   ▷ Function call to check if vertex is cut vertex
                append($cutVertices, i$)
            end if
        end if
    end for
    for $i$ in $cutVertices$ do                              ▷ Printing cut vertices
        print $i$
    end for
end function
```

Correctness of Algorithm

The algorithm correctly finds all the cut vertices by systematically evaluating each vertex in the graph and determining whether removing that vertex would disconnect the source vertex s from the target vertex t. Here's a brief explanation of why the algorithm works:

1. **DFS Traversal**: The algorithm uses Depth-First Search (DFS) to traverse the graph from the source vertex s to the target vertex t. During this traversal, it marks visited vertices to keep track of the path taken.
2. **Cut Vertex Identification**: For each vertex v in the graph (excluding s and t), the algorithm temporarily "removes" v from the graph (skips calling DFS on it) and performs a DFS traversal from s to t. If t is still reachable after removing v, it means that v is not a cut vertex because there exists an alternate path from s to t that does not include v.
3. **Recording Cut Vertices**: If t becomes unreachable after removing v, it implies that v is a cut vertex because removing it from the graph disconnects the path from s to t. In this case, the algorithm records v as a cut vertex.
4. **Returning Cut Vertices**: After evaluating all vertices (excluding s and t), the algorithm returns the list of recorded cut vertices.

By systematically evaluating each vertex's impact on the connectivity between s and t, the algorithm ensures that it correctly identifies all the vertices that, if removed, would disconnect s from t. Therefore, the algorithm correctly finds all the cut vertices in the graph.