Theory Assignment-4: ADA Winter-2024

Ritika Thakur (2022408)   Sidhartha Garg (2022499)

1 Preprocessing

In the preprocessing phase, we read the input graph and perform topological sorting to obtain the ordering of nodes.

2 Algorithm Description

The algorithm consists of the following steps:

1. Read input for the number of nodes n, the number of edges m, and the start and end nodes.
2. Initialize an adjacency matrix representation of the graph.
3. Read the edges of the graph and update the adjacency matrix.
4. Perform topological sort using Depth First Search (DFS) to obtain the ordering of nodes.
5. Iterate through the nodes in reverse topological order, excluding the start and end nodes.
6. For each intermediate node, check if removing it disconnects the path between the start and end nodes using Breadth First Search (BFS).
7. If removing a node disconnects the path between start and end nodes then it is a cut vertex.

3 Proof of Correctness

Let's denote the start node as s and the end node as e. We aim to prove the correctness of the algorithm by showing that it correctly identifies cut vertices, i.e., nodes whose removal disconnects the path from s to e.

1. **Topological Ordering:** The algorithm ensures that the topological ordering is obtained correctly using DFS. This ordering guarantees that if there exists a path from s to e, s will appear before e in the ordering.

2. **Path Checking:** For each intermediate node i, the algorithm checks if removing i disconnects the path from s to e. This is achieved by performing BFS while excluding node i. If BFS fails to reach e from s without passing through i, then i is a cut vertex.

3. **Correctness:** Since the algorithm exhaustively checks all intermediate nodes and correctly identifies cut vertices, it accurately determines whether there exists a node whose removal disconnects the path from s to e.

4 Complexity Analysis

4.1 Time Complexity

Depth-First Search (DFS) for Topological Sorting:
During the topological sorting phase, we perform a DFS traversal on the graph. In each DFS traversal, we visit each vertex and each edge once. Therefore, the time complexity of DFS is $O(V + E)$, where V is the number of vertices and E is the number of edges.

Breadth-First Search (BFS) for Path Checking:
In the path checking phase, we perform a BFS traversal to check if removing an intermediate node disconnects the path between the start and end nodes. For each of the $(V - 2)$ intermediate nodes, we perform a BFS traversal, which has a time complexity of $O(V + E)$ each. Therefore, the total time complexity for path checking is $O((V - 2) \times (V + E)) = O(V(V + E)) = O(V^2 + VE)$.

Overall Time Complexity:
Adding the time complexities of topological sort and path checking, we get $O(V + E + V^2 + VE)$. Thus, the overall time complexity of the algorithm is $O(V^2 + VE) = O(V(V + E))$.

Matching the Algorithm's Running Time:
The algorithm traverses each vertex and each edge once during both DFS and BFS. In DFS, we explore each vertex and its outgoing edges, ensuring that we correctly order the vertices topologically. In BFS, we explore each vertex and its outgoing edges to determine path connectivity. By performing these traversals efficiently, our algorithm correctly identifies cut vertices and maintains a time complexity of $O(V(V + E))$.

Additional Consideration:
The step of constructing the adjacency matrix has a time complexity of $O(V^2)$. However, for large graphs, this is often negligible compared to the complexities of DFS and BFS. Therefore, it does not significantly affect the overall time complexity of the algorithm.

4.2 Space Complexity

The space complexity of the algorithm is $O(V^2)$ due to the adjacency matrix representation of the graph and other auxiliary data structures used during DFS and BFS.

5 Pseudocode

Below is the pseudocode of the algorithm implemented using adjacency matrix representation of graph:-

Algorithm 1 Finding Cut Vertices

```
function DFS($node, graph, visited, ordering$)
    $visited[node] \leftarrow 1$
    for $i$ from 1 to $n$ do
        if $graph[node][i] == 1$ and $visited[i] == 0$ then
            DFS($i, graph, visited, ordering$)
        end if
    end for
    $ordering$.push_back($node$)
end function
function CheckPath($start, end, node, visited, graph$)
    $q \leftarrow$ new Queue
    $q$.push($start$)
    $visited[start] \leftarrow 1$
    while $q$ is not empty do
        $curr \leftarrow q$.front()
        $q$.pop()
        if $curr == end$ then
            return True
        end if
        for $i$ from 1 to $n$ do
            if $graph[curr][i] == 1$ and $visited[i] == 0$ and $i \neq node$ then
                $q$.push($i$)
                $visited[i] \leftarrow 1$
            end if
        end for
    end while
    return False
end function
function Main
    $n, m \leftarrow$ Read input for number of nodes and edges
    $start, end \leftarrow$ Read input for start and end nodes
    $graph \leftarrow$ Initialize graph adjacency matrix with zeros
    for $i$ from 1 to $m$ do
        $u, v \leftarrow$ Read input for edge $(u, v)$
        $graph[u][v] \leftarrow 1$
    end for
    $visited \leftarrow$ Initialize visited array with zeros
    $ordering \leftarrow$ Empty vector for ordering of nodes
    for $i$ from 1 to $n$ do
        if $visited[i] == 0$ then
            DFS($i, graph, visited, ordering$)
        end if
    end for
    $check \leftarrow 0$
    for $i$ from $size(ordering)$ down to 1 do
        if $ordering[i] \neq start$ and $ordering[i] \neq end$ then
            $newVisited \leftarrow$ Initialize new visited array with zeros
            if !CheckPath($start, end, ordering[i], newVisited, graph$) then
                Print("$Removing\ node$ " $+ ordering[i] +$ " $will\ disconnect\ the\ path\ from$ " $+ start +$ " $to$ " $+ end$")
                $check \leftarrow 1$
            end if
        end if
    end for
    if $check == 0$ then
        Print("$No\ node\ can\ be\ removed\ to\ disconnect\ the\ path\ from$ " $+ start +$ " $to$ " $+ end$")
    end if
end function
```