Theory Assignment-4: ADA Winter-2024

Nikhil Kumar (2022322) Nikhil (2022321)

29 March 2024

1 Algorithm Description

The basic idea behind this algorithm is to iteratively remove each vertex from the graph and check if there is still a path from the start vertex (s) to the target vertex (t). If there exists a path from vertex s to vertex t, even after removing a vertex u, then vertex u cannot be a (s,t) cut vertex. Conversely, if there does not exist a path from vertex s to vertex t, then we can conclude that vertex u is a (s,t) cut vertex.
The algorithm utilizes a depth-first search (DFS) approach to traverse the graph and efficiently determine the presence of paths.

1.1 Main Steps of the Algorithm

1. The input for the graph is taken as an adjacency list.
2. After that, we make an initial call to the function `find_cut_vertices()` with arguments as the adjacency list and our s, t vertices.
3. The function `find_cut_vertices()` identifies the cut vertices in the graph. Now it starts traversing the adjacency list of the graph, and for each vertex, say u, it creates a modified graph by removing u.
4. This part of removing the vertex u from the graph can easily be carried out by just removing the adjacency entry of the vertex u.
5. Now, it performs a DFS traversal of the modified graph using another helper function `dfs()`. If there is no path from vertex s to vertex t in the modified graph, then vertex u is identified as a cut vertex and added to the set of cut vertices.
6. Now, when this function calls the function `dfs()`, then it just simply performs a depth-first search (DFS) traversal of the graph starting from a given start vertex and ending at an end vertex.
7. After performing DFS traversal, It returns True if there exists a path from the start vertex to the end vertex, and False otherwise.
8. Now, when the function `dfs()` returns, if the returned value is True, then we store the vertex u as the cut vertex.

2 Complexity Analysis

2.1 Time Complexity

To analyze the time complexity of the `find_cut_vertices` function:

1. **Outer Loop (Iterating over Vertices):** The outer loop iterates over each vertex in the graph, excluding the start (s) and end (t) vertices. This loop runs once for each vertex, so it has a time complexity of $O(V)$, where V is the number of vertices in the graph.

2. **Inner Operations (Removing Vertex and DFS):**

   - Removing a vertex from the graph involves removing the adjacency entry of that vertex. In the worst case, it would traverse the whole adjacency list. This operation takes $O(V)$ time because it involves traversing the entire graph and then removing it.
   - The DFS function is called to check if there exists a path from s to t in the modified graph. The time complexity of DFS of the graph is $O(V + E)$, where V is the number of vertices and E is the number of edges.

3. **Total Time Complexity:**

   - The outer loop runs $O(V)$ times.
   - Inside the loop, the removal of the vertex takes $O(V)$ time.
   - The DFS operation inside the loop takes $O(V + E)$ time.
   - Since these operations are nested, we multiply their time complexities.

So, the overall time complexity of the `find_cut_vertices` function can be expressed as $O(V \cdot (V + E))$.

2.2 Space Complexity

To analyze the space complexity of the `find_cut_vertices` function:

1. **Graph Representation:** The input graph is represented using an unordered map data structure. In C++, the space complexity of an unordered map is $O(V + E)$, where V is the number of vertices and E is the number of edges.
2. **Visited Set:** Inside the DFS function, a set named `visited` is used to keep track of visited vertices. The size of this set can grow up to the number of vertices in the graph, so its space complexity is $O(V)$.
3. **Recursion Stack:** The space complexity of the recursion stack in the DFS function depends on the maximum depth of recursion, which is the length of the longest path from the start vertex to any other vertex in the graph. In the worst case, this depth can be $O(V)$, where V is the number of vertices in the graph.
4. **Copied Graphs:** Within the outer loop, a copy of the graph is created by using the `copy()` method. The space complexity of this copy operation is also $O(V + E)$, similar to the original graph.
5. **Overall Space Complexity:** Combining all these factors, the overall space complexity of the `find_cut_vertices` function can be expressed as $O(V + E) + O(V) + O(V) + O(V + E)$, which simplifies to $O(V + E)$.

Therefore, the space complexity of the `find_cut_vertices` function is $O(V + E)$, where V is the number of vertices and E is the number of edges in the graph.

3 Pseudocode

```
1: function DFS(graph, start, end, visited)
2:     if start == end then
3:         return true
4:     end if
5:     visited.insert(start)
6:     for neighbor in graph[start] do
7:         if visited.find(neighbor) == visited.end() then
8:             if DFS(graph, neighbor, end, visited) then
9:                 return true
10:            end if
11:        end if
12:    end for
13:    return false
14: end function
```

```
1: function FIND_CUT_VERTICES(graph, s, t)
2:     cut_vertices ← {}
3:     for auto & vertex : graph do
4:         if vertex.first ≠ s and vertex.first ≠ t then
5:             auto removed_graph = graph;
6:             removed_graph.erase(vertex.first);
7:             unordered_set < char > visited;
8:             if not DFS(removed_graph, s, t, visited) then
9:                 cut_vertices.insert(vertex.first);
10:            end if
11:        end if
12:    end for
13:    return cut_vertices
14: end function
```

4 Explanation of correctness of algorithm

The `find_cut_vertices` function traverses each vertex in the graph, excluding the start and end vertices. For each vertex, it creates a copy of the graph with the current vertex removed.
Then, it performs a depth-first search (DFS) from the start vertex to the end vertex on the modified graph. If no path is found, indicating that the removal of the vertex disconnects the start from the end, the vertex is identified as a cut vertex.

This process ensures that every vertex (except the start and end vertices) is considered for being a cut vertex. By individually removing each vertex and checking for connectivity between the start and end vertices, the function accurately identifies all cut vertices without falsely labeling non-cut vertices.
The use of DFS guarantees that only vertices whose removal results in the disconnection of the start and end vertices are identified as cut vertices, ensuring the correctness of the function.