Complete Analysis of the task

The problem to solve for this task was to implement an algorithm to find the connections of different social media users and return the largest connected groups. The connection of the users can be mapped out as a directed graph with nodes. So for this task, we had to make a code to navigate a directed graph and find the largest group inside the graph where you can reach all adjacent vertices from a specific node. Now, when conducting research on this task and how to solve it I had found two primary algorithms that are used for this specific problem. One was called Kosaraju's algorithm and the other was Tarjan's algorithm. The one I decided to use was Kosaraju's algorithm and I will explain why below.

What algorithm did you use? Why?

As mentioned earlier the chosen algorithm I went for was Kosaraju's algorithm. It is a common algorithm that is used to find the largest group in a directed graph. The first step of the algorithm is to perform a depth-first search and save the order of the nodes based on their finishing times. Then, we will transpose the graph or in other words reverse the order of all the directions of the edges. And perform the depth-first search one more time but this time on the transposed graph. And lastly, we will record the size of each of the strongly connected groups while navigating the graph again.

What other algorithms apply to this task?

The other algorithm which also is the one that is usually slightly preferred is the Kosaraju's algorithm. The reason why it is usually preferred is because it finds the largest groups in only one depth first search. Making it more efficient as we do not have to transpose the graph. So why did I choose the other algorithm? It is because this approach even though it's more efficient it is harder to implement. And the algorithm I chose is also easier to understand and therefore I felt more comfortable choosing this algorithm over it since our problem is not a large scale problem. So you would not notice the difference in efficiency as you would on a larger scale.

In what ways does the algorithm you used differ from these other algorithms?

The main difference from the algorithm that I used is that two depth first searches are required to find the largest groups instead of one. But it is simpler to understand and implement.

How does the code work? A detailed explanation of the functions made.

The code includes comments to fully explain each function made

Complexity analysis of the algorithm.

The time complexity is O(n). This is because we linearly navigate the graph. Also, we utilize data structures such as stacks which also follow a linear approach.

Code:

```
#include <iostream>
#include <vector>
#include <stack>
#include <list>

using namespace std;

class Graph {
    int vertNum;

    // We are going to create a pointer to an array containing the adjacency list
    list<int>* adj;

public:

    // In our constructer we are declaring our functions
    Graph(int numberOfVertices);
    void createEdge(int v, int w);
    void orderVertices(int v, bool visited[], stack<int>& Stack);
    void findGroups(int v, bool visited[], int& size);
    void largestGroup();
    Graph transpose();
```

```
};


Graph::Graph(int numberOfVertices) {
    this->vertNum = numberOfVertices;
    adj = new list<int>[numberOfVertices];
}


// This function utulizes depth first search recursively from vertex v. Any adjacent vertex is explored and marked as visited once done. And as we are navigating the adjacent edges we are keeping a count to determine at the end the largest group
void Graph::findGroups(int v, bool visited[], int& size) {
    // Marking the current edge as visited
    visited[v] = true;
    // Keeping a count of the size
    size++;


    // Recursive call for all adjacent vertixes that have not been visited yet
    for (auto i = adj[v].begin(); i != adj[v].end(); ++i)
        if (!visited[*i])
            findGroups(*i, visited, size);
}


// This is our transpose funciton. This will reverse the order of every edge.
Graph Graph::transpose() {
    Graph g(vertNum);
    for (int v = 0; v < vertNum; v++) {
        for (auto i = adj[v].begin(); i != adj[v].end(); ++i) {
            g.adj[*i].push_back(v);
```

```
        }
    }
    return g;
}

// This function will allow us to manually add edges to our graph later in the main function
void Graph::createEdge(int v, int w) {
    adj[v].push_back(w);
}

// Here we are ordering our vertices in order of their finishing time
void Graph::orderVertices(int v, bool visited[], stack<int>& Stack) {
    // marking our vertix we are currently on as visited
    visited[v] = true;

    // same recursive call to go through all adjacent vertices that have not been visited yet
    for (auto i = adj[v].begin(); i != adj[v].end(); ++i)
        if (!visited[*i])
            orderVertices(*i, visited, Stack);

    // Pusing the vertices that we were able to reach into a stack
    Stack.push(v);
}

// As the name implies this function is the find the largest aka the strongest connection groups.
void Graph::largestGroup() {
    stack<int> Stack;
```

```
    // Mark all vertices as not visited for first depth first search. This will help us keep track of the visited verteces instead of having to go back later to check
    bool* visited = new bool[vertNum];
    for (int i = 0; i < vertNum; i++)
        visited[i] = false;

    // Based of their finishing times we will push them to a stack. So our last visited vertex will be at the top of the stack.
    for (int i = 0; i < vertNum; i++)
        if (!visited[i])
            orderVertices(i, visited, Stack);

    // Now we will transpose the graph. In other words reverse the order
    Graph gr = transpose();

    // Same thing again here we will mark all the verteces as unvisited again.
    for (int i = 0; i < vertNum; i++)
        visited[i] = false;

    // Now we will navigate the stack that we have previously created. And start to pop every vertex one by one. But everytime we will check if it has been visited or not.
    while (!Stack.empty()) {
        int v = Stack.top();
        Stack.pop();

        // If we find one vertex that has not been visited yet we will call the findGroups() function to perform the depth first search again but this time on the transpose graph and count the groups
        // And at the end we print out the sizes of the group
```

```
        if (!visited[v]) {
            int size = 0;
            gr.findGroups(v, visited, size);
            cout << "Size of group is " << size << endl;
        }
    }

    // Freeing up the memory after we are finished
    delete[] visited;
}

// Main function
int main() {
    cout << "Initializing Kosaraju's algorithm to find the largest groups based of the given graph: "
<< endl;
    // Calling the graph function
    Graph g(7);

    g.createEdge(1, 0);
    g.createEdge(0, 2);
    g.createEdge(2, 1);
    g.createEdge(0, 3);
    g.createEdge(3, 0);
    g.createEdge(4, 3);
    g.createEdge(3, 4);

    // Calling our function to print all strongest connected groups
```

```
    g.largestGroup();


}
```