Algorithm Analysis
CSI 5301

Assignment 5.1: The UN Problem
Due: Mon June 13, 2022 11:59pm
Submitted by: Pawan Bhandari

Description:

Read from a input file that contains the following information:

Input starts with a positive integer, $1 \leq N \leq 100$, the number of members in the meeting. This is followed by N lines, each line describing a member. Each of these N lines starts with the member's name (which is distinct), then the language(s) that the member speaks (this will only be a single language), then a list of 0 to 20 additional languages the member understands but doesn't speak. All members understand the language they speak. All members and language names are sequences of 1 to 15 letters (a-z and A-Z), numbers, and hyphens. Member names and languages are separated by single spaces.

For example:
Ian Italian French Spanish
In this case, the name would be "Ian," who speaks "Italian" and understands "French" and "Spanish." As mentioned previously, there will always be a single language that is spoken (the one right after the name). Any additional languages they understand will be after that first spoken language, although they might not understand any other language.

After processing the input file, print the minimum number of members that will be required to leave so that all remaining members can converse with each other in a meeting.

Data Structure:

Graph
vector<string>, vector<Member>, stack<Member>
set<string>, bool[ ]

Algorithm:

1) Read input.txt to populate memberVector and find number of members

readInputFile (vector<Member> *memberVector, string inputFileName)

start
numberOfMembers = total number of members
member = representation of each member as custom member object

open ifstream on inputFileName (input.txt in this case)
read first line and set it as numberOfMembers
read other lines and create a member form each line and push it to memberVector
end

2) Create a Adjacency list representation Graph with members as vertices and add a directional edge from member1 to member2 if member2 can understand a language that member1 speaks

createMemberGraph (vector<Member> *memberVector, MemberGraph *UNMemberGraph, ofstream &outputData)

start

loop through each Member, say m1 in the memberVector and add an edge to another Member, say m2 if m1.speaks() = m2.speaks() or m2.understands() contains m1.speaks()

Also add a reverse edge on a new adjacency list so that it can be used later for a second DFS while implementing Kosaraju's Algorithm

addEdge(Member mSpeaks, Member mUnderstands) in MemberGraph adds an edge from mSpeaks to member mUnderstands

end

3) Apply Kosaraju's algorithm on the MemberGraph to find a list of strongly connected members (i.e. group of members who can converse with each other)

MemberGraph::findSCC()

start

initialize a vector to hold a vector of strongly connected members
initialize a stack of Member to hold members
initialize a boolean array to keep track if visited members

do a first Depth-First-Search on MemberGraph

fill the stack<Members> with members as they are finished exploring during DFS

do a second Depth-First-Search on reversed MemberGraph (created in step 2) staring from the top member of the stack created during first Depth-First-Search

save the list of members from each run of second Depth-First-Search, this will return a vector with each element as a vector of strongly connected members

end

4) Calculate minimum number of members required to leave

start

maxSccMembersSize = the size of largest SCC of members from step 3.
numberOfMembers = total number of members (calculated in step 1).

membersRequiredToLeave = *minimum number of members required to leave.*
membersRequiredToLeave = *numberOfMembers* – *maxSccMembersSize*
print membersRequiredToLeave as output to the terminal.

end

Analysis:

Input N	int numberOfMembers Vector<Members> memberVector (will be implicitly derived from input.txt)
Basic Operation	Depth-First-Search (DFS) on a Graph Kosaraju's Algorithm Implementation Vector operations Recursion
Recurrence relation	
Run-time analysis (Big O)	$O(V^2)$ for adding edges in Graph $O(V+E)$ for determining output by implementing Kosaraju's Algorithm

Run-time Analysis

Following are the significant steps that will influence the runtime of this program

V = number of members/vertices & E = Number of edges in MemberGraph

1) Reading the input file and adding edges to MemberGraph takes $O(V^2)$ time as it loops through the list of members for each member to add an edge on MemberGraph.
2) Creating a reversed Graph simultaneously as we add edges takes $O(V^2)$ time.
3) Running a DFS on Initial MemberGraph takes $O(V+E)$ time.
4) Running a second DFS on reversed MemberGraph also takes $O(V+E)$ time.

In this case, the overall running time of program can be expressed as $O(V^2)$
Run time of the Kosaraju's Algorithm to find strongly connected members is $O(V+E)$

Worst Case Analysis:

Maximum possible edges for a dense graph is V^2 . So if we were to improvise on processing the input file, the worst runtime of the Algorithm (Kosaraju's) is $O(V+V^2)$ i.e. $O(V^2)$

Best Case Analysis:

In case of a sparse graph, (fewer edges) the runtime will be of $O(V+E)$ which is the best case.