**Istanbul Technical University**

**Faculty of Computer and Informatics**



**BLG3356 Analysis of Algorithms 2**

**Project 1 Report**

**Cem Yusuf Aydoğdu**

**150120251**

# a) Finding the shortest path(s) between two nodes

For finding shortest paths from a starting node to an end node, a modified version of breath first search algorithm is used.

In this modified version, a counter is used for counting paths and a variable is used to check path lenghts. While considering each edge incident to edge , if the node is the end node (whether it is discovered or not) and if the path lenght to node is the shortest possible lenght to that node, the path counter is incremented as one.

# b) Computing betweenness of each edge

In computing betweenness of each edge, paths from each node to another nodes are found first. Then, for each these paths, counters for each found edge are incremented by one. After that, counters for each edge is printed to the screen.

# c) Testing if the graph is strongly connected or not

Testing the directed graph for strong connectivity also depends on breath first searching.

First, a random node in the graph is selected as starting point of the breath first search. Number of reached nodes are calculated in the search. Then, all edges of the graph are reversed, which results a new graph . Breath first search is performed again with the same node in , again the number of reached nodes are calculated. If both numbers of reached nodes are equal to remaining number of nodes in the graph, then it means graph is strongly connected.

## Pseudocode for a)

**findNumberOfShortestPaths**(start, end):

Set discovered[start]=true,

Set discovered[u]=false for each node u≠start

Initialize layer L[0] with element start

Set layer counter i=0

Set path counter p=0,

Set path lenght len=∞

While L[i] is not empyt:

Initialize an empty layer L[i+1]

For each edge (u,v) of node u in L[i]:

If discovered[v]=false:

Set discovered[v]=true

Add v to layer L[i+1]

Endif

If v=end and i+1≤len:

Increment path counter p by one

Increment path length len by i+1

Endif

Endforeach

Increment layer counter i by one

Endwhile

Return path counter p

# Complexity

This function is a modified version of breath first search. However, modifications have no effect in overall complexity, Since the graph is represented as adjacency list, the complexity of this function is where and .

## Pseudocodes for b)

**getPath**():

Set ,

Set for each node

Set array for each node // for constructing the path

Initialize layer with element

Set layer counter

Set bool

While :

Initialize an empty layer

For each edge of node in :

If :

Set

Add to layer

Set

Endif

If :

Set

Endif

Endforeach

Increment layer counter by one

Endwhile

Initialize empty stack // Construct the path with a stack

Push to stack

Set

While

Push to

Set

Endwhile

Push to // Stack has the path now

Initialize empty list

While is not empty: // Reverse the stack with by pulling

Pull top element in to list

Endwhile

Return list

# Complexity

This function is also a modified version of breath first search. In the first part of the function (first while loop), complexity is determined by . In the second part, pushing or pulling values to/from the stack is upper bounded by number of edges, .

So the overall complexity for this function .

**findEdges**():

Initialize empty edge list

For each node :

For each edge of :

If the edge is not in the list:

Add the edge to the list

Endif

Endforeach

Endforeach

# Complexity

This procedure contains two loops. Total complexity of these nested loops are , where and .

**computeBetweenness**():

Set all edges in the graph // using findEdges function

Set edge counter for all edges

Initialize an empyt list

For each node :

For each node starting from next node of , :

Get path from to //using getPath function

Add the to the list

Endforeach

Endfor

For each path in

For each edge in the path

Increment for corresponding edge by one

Endforeach

Endfor

# Complexity

First, complexity of finding all edges is. After that, completixy of the nested loop is , and getting shortest paths between nodes and is bounded by , which results as . Finally, checking each edge in paths requires . Total complexity of this function is bounded by .

## Pseudocodes for c)

**reverseEdges**():

Initialize empty graph

For each node :

For each edge of :

Add reverse of the edge to

Endforeach

Endforeach

# Complexity

All edges of all nodes must be reversed (reverse operation is ), so the total complexity is , where and .

**findNumberOfShortestPaths**(start):

Set ,

Set for each node

Initialize layer with element start

Set layer counter

Set reached counter ,

While is not empyt:

Initialize an empty layer

For each edge of node u in :

If

Set

Increment counter by one

Add to layer

Endif

Endforeach

Increment layer counter by one

Endwhile

Return counter

# Complexity

This function is also a modified version of breath first search, with a counter for number of reached nodes added. Complexity of this function is where and .

**checkStrongConnectivity**():

Set node as a random node

Set as the reverse of current graph // using reverseEdges()

Set reach count

Set reach count

If and where

// Strongly connected

Else

// Not strongly connected

Endif

# Complexity

Reversing the graph costs , and calculating reach counts for both and costs . Total complexity is .

## Implementation Details

In the implementation, file names for graphs are taken in the main function of the code. In order to represent the graph with an adjacency list, *vector< list<int> >* data type from STL library is used (defined in *graph.h*  header file).

There are three classes, and since there are two types of graphs(undirected and directed) two classes are used to represent these types, *Graph\_undirected* and *Graph\_directed.* These two classes are child of a parent class *Graph* which contains common methods and data structures. UML class diagram is given below.

