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 (u, v) incident to edge u, if the node v is the end node (whether it is discovered or not) and if the path lenght to node v 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$s$ in the graph$G$ 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 $G^{reverse}$ .Breath first search is performed again with the same node$s$ in $G^{reverse}$, 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
```

$O(n + m)$

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 $O(m + n)$ where $n = |V|$ and $m = |E|$.

Pseudocodes for b)

getPath($start,\ end$):

Set $discovered[start] = true$,

Set $discovered[u] = false$ for each node $u \neq start$

Set array $previous[u] = -1$ for each node $u \in V$ // for constructing the path

Initialize layer $L[0]$ with element $start$

Set layer counter $i = 0$

Set bool $found = false$

While $found \neq true$:

  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 $L[i+1]$

      Set $previous[v] = u$ $O(n+m)$

    Endif

    If $v = end$:

      Set $found = true$

    Endif

  Endforeach

  Increment layer counter i by one

Endwhile

Initialize empty stack s // Construct the path with a stack

Push end to stack s

Set $i = previous[end]$

While $i \neq start$:

  Push p to s $O(n)$

  Set $i = previous[i]$

Endwhile

Push $start$ to s // Stack has the path now

Initialize empty list $path$

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

  Pull top element in s to list $path$ $O(n)$

Endwhile

Return list $path$

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 $O(n+m)$. In the second part, pushing or pulling values to/from the stack is upper bounded by number of edges, $O(n)$. So the overall complexity for this function$O(n+m)$.

findEdges():

Initialize empty edge list E

For each node $u \in V$:

For each edge (u, v) of u:

If the edge is not in the list:

Add the edge to the list

Endif

Endforeach

Endforeach

$O(nm)$

Complexity

This procedure contains two loops. Total complexity of these nested loops are $O(nm)$, where $n = |V|$ and $m = |E|$.

computeBetweenness():

Set $edges$ = all edges in the graph $O(nm)$ // using findEdges function

Set edge counter $count = 0$ for all edges

Initialize an empyt list $paths$

For each node $u \in V$:

For each node $v \in V$ starting from next node of u,$v \neq u$:

$O(n+m)$ Get path$p$ from u to v //using getPath function

Add the p to the list $paths$ $O(n^2(n+m))$

Endforeach

Endfor

For each path p in $paths$:

For each edge $e \in edges$ in the path p:

Increment $count$ for corresponding edge by one $O(nm)$

Endforeach

Endfor

Complexity

First, complexity of finding all edges is$O(nm)$. After that, completixy of the nested loopis $O(n^2)$, and getting shortest paths between nodes u and v is bounded by $O(n+m)$, which results as$O(n^2(n+m))$. Finally, checking each edge in paths requires $O(nm)$. Total complexity of this function is bounded by$O(n^3)$.

Pseudocodes for c)

reverseEdges():

Initialize empty graphG^{rev}

For each node $u \in V$:

For each edge (u, v) of u:

Add reverse of the edge to G^{rev} $O(nm)$

Endforeach

Endforeach

Complexity

All edges of all nodes must be reversed (reverse operation is $O(1)$), so the total complexity is$O(nm)$, where $n = |V|$ and $m = |E|$.

findNumberOfShortestPaths(start):

Set $discovered[start] = true$,

Set $discovered[u] = false$ for each node $u \neq start$

Initialize layer $L[0]$ with element start

Set layer counter $i = 0$

Set reached counter $c = 0$,

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 $discovered[v] = true$

Increment counter c by one $O(n + m)$

Add v to layer $L[i + 1]$

Endif

Endforeach

Increment layer counter i by one

Endwhile

Return counter c

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 $O(m + n)$ where $n = |V|$ and $m = |E|$.

checkStrongConnectivity():

Set node u as a random node $\in V$

Set G^{rev} as the reverse of current graph $O(nm)$ // using reverseEdges()

Set reach count $c1 = findNumberOfNodesReached(G, u)$

Set reach count $c2 = findNumberOfNodesReached(G^{rev}, u)$

$O(n + m)$

If $c1 = n - 1$ and $c2 = n - 1$ where $n = |V|$

// Strongly connected

Else

// Not strongly connected

Endif

Complexity

Reversing the graph costs $O(nm)$, and calculating reach counts for both G and G^{rev} costs $O(m + n)$. Total complexity is $O(nm + n + m) = O(nm)$.

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.