HW5

Shane Drafahl

31 October,2017

1. This algorithm goes over every node in the graph and checks the neighbors of the neighbors and populates a adjacency list for nodes of distance 2 from each other. After this the adjacency list is then set to what the new adjacency list is used.

```
// Given a graph G createG2 converts G to G2 graph
function createG2(G) {
    foreachNode(G as N) {
        foreachNeighbor(N as N1) {
            foreachNeighbor(N1 as N2) {
                append(N.N2, N2);
            }
        }
    }
    foreachNode(G as N) {
        N.N = N.N2;
    }
}

Node {
    N[] // adjacency list
    N2[] // adjacency list for G squared
}

Graph {
    N[] // List of nodes
}
```

This algorithm runs in $O(nm)$ because the first for loop goes over n vertices and the second for loop that goes over every vertice. That means the two for loops is just $O(n + m)$. There is a fourth for loop though that also goes over vertices so it is $O(m(n + m)) = O(mn + m^2)$. $m * n$ is the biggest factor because m at most can be $m = n^2$ so therefore it is $O(mn)$.

```
// Given a adjacency matrix where (x, y) and x is the domain and the y is th
// there is a function between different vertices to other vertices.
function createG2(M) { // suppose that M is a adjacency matrix.
       M2 // new Matrix of equal size and width of M
       x = 0
       for(x in range(M.height)) {
           y = 0
           for(y in range(M.width)) {
               if(M[x][y] == 1) {
                   a = 0
                   for(a in range(M.width)) {
                       if(M[y][a] == 1) {
                           M2[x][a] = 1
                       }
                   }
               }
           }
       }
       return M2
}
```

This algorithm goes over every index of the matrix which is $O(n^2)$ where n is the number of vertices. If it finds a connection it has another for loop

an then updates the initial row. Worst case scenario this is a fully connected graph so $O(n^3)$ best case the graph has no edges or connections it would be $O(n^2)$.

2. For this problem I will use Kosaraju's algorithm to see if there is a vertice with equal number of nodes behind it as in front of it.

```
function hasCenter(G) { // suppose G is the Grahp
       ////////////////////////////////////////// Kosaraju algorithm
       L // list of strongly connected nodes with only a single node
       S = new Stack // new stack
       forEachNode(G as N) {
           if(!N.visited) {
               N.visited = true
               dfs(N, S)
           }
       }
       while(!isEmpty(S)) {
           Node = pop(S)
           if(Node.Root == null) {
               dfsStronglyConnected(Node, Node, L)
           }
       }
       //////////////////////////////////////////
       forEach(L as STN) { // each element in L as STN or strongly connect n
           size = countRightNodes(STN) - 1
           sizeOfGraph = size(G.L)
           sizeOfGraph = (sizeOfGraph - 1)/2
           if(sizeOfGraph == size) {
               return STN
           }
       }
       return false
}
```

```
// recursive algorithm that creates the strongly connected vertices
// Root is a Node that is chosen that will represent the strongly connected
function dfsStronglyConnected(Node, Root, L) {
   Node.Root = Root
   Root.quant++
   foreachNeighbor(Node as N) {
       dfsStronglyConnected(N, Root, L)
   }
   if(Root.quant == 1) {
       append(L, Node) // adds the Node to the list L
   }
}

function countRightNodes(N) {
    N.visited = false
    if(hashNeighbors(N)) {
        foreachNeighbor(N as N1) {
            if(N1.visit = true) {
               return 1 + countRightNodes(N1)
            }
        }
    }
}

// N is a node
// S is a stack
function dfs(N, S) {
    if(hashNeighbors(N)) {
        foreachNeighbor(N as N1) {
            dfs(N1, S)
        }
    } else {
        push(S, N) // pushes N onto stack S
    }
}

Node {
   Root // root of the node
   quant = 0 // number of nodes it represents
   visited = false
```

```
   Neighbors[] // adjacent neighbors
}

Graph {
    L // list of nodes
}
```

The algorithm above first uses Kosaraju's algorithm to find all the strongly connected components in the graph and connect each one to a root that contains the number of nodes in each strongly connected component. Once this is done I keep track of the strongly connected nodes with only a single node because v in the algorithm will be a single strongly connected component with a single node because the set of From and To cannot overlap. I then itterate over all strongly connected components with single nodes and using DFS find the number of nodes to the right. If the graph has a center then there should be exactly k nodes to the right of v. So Once I find the total number of nodes to the right I take the total number of nodes for the whole graph subtract 1 and divided by 2 to get k. If the number of nodes to the right of v equal k then I know the graph has a center.

If k is the center of the graph then it cannot be part of the TO or FROM set so k cannot have a path from k to k. Meaning k is not in a cycle. So if we found all the strongly connected components k would be in a strongly component with itself. So if we use Kosaraju's algorithm we will find all the strongly connected components. This algorithm stores a list of strongly connected components with a single node. I then use DFS on each of these nodes to find the quantity of nodes and subtract 1 to only find the quantity of vertices and not the center vertice adjacent to the node. If the node is the root of a tree and it finds the number of nodes under the root is equal to k then the root is the center of the graph. If it is k that means there must be k vertices that are connected to the center vertice with no path to them.

Kosaraju's algorithm runs in linear time so $O(n + E)$. The forloop after the algorithm runs through every strongly connected component. Worst case

every node could be strongly connected so it would be essentialy a linked list. In this case worst case be k iterations. So it would be $O(n+E+\frac{(n^2+n)}{2})$ or $O(n^2+E)$ worst case. Best case if there is only one strongly connected component then $O(n+E)$. For every case $O(n+E+f)$ if f is the number of strongly connected nodes.

3. Is there a vertex $u \in V$ such if we perform DFS on G starting at u , the vertex v will be a leaf node in the resulting DFS tree?

No, a leaf node in the recursion tree for DFS can only happen if a node or a vertice is reached without calling any adjacent vertices. A bridge node cannot be the leaf node or else one side of the graph would be unvisited.

Because this algorithm must be in $O(n+m)$ we know it must be DFS based. From the previous question we know it cannot be a leaf node so we simply just need to do a DFS search and stop once we reach a leaf node and return it.

```
function NonBridgeDFS(G) {
    return search(G.adj[0])
}

// N is a vertice or a node
function search(N) {
    N.visited = true
    leaf = true
    foreachNeighbor(N as N1) {
        if(N1.visited == false) {
            leaf = false
            return search(N1)
        }
    }
    if(leaf) {
```

```
        return N
    }
}

Node {
    visited = false
}
```

This algorithm is correct because we know that a leaf node cannot be a bridge node. A leaf node is one where when search is used on the node it cannot find any more adjacent Nodes that have not been visited before. The first line in NonBridgeDFS picks an arbitrary node in the graph. I then use search() on that node. In search() N is marked as visited and a boolean leaf is set to true. Then there is a forloop over all the neighbors of the node. If it finds a single adjacent node that has not been visited before it changes leaf to false. If there is a adjacent node that has not been visited search is used on the adjacent node. Essentialy this is DFS. search will reach the base case on either side and not the bridge vertice. If we use proof by contradiction and assume search will reach the base case on the bridge vertice and return the vertice. This would mean that when visit() is called on the bridge all adjacent vertices are visited. This would mean that there would be another connection between two sets vertices. This is a contradiction because that would mean there would be no bridge vertice because if we removed the assumed bridge vertice the graph would not become disconected. So therefore this algorithm will return any vertice that is not a bridge vertice.

This algorithm is $O(n+m)$ where n is the number of vertices and m is the number of edges. The recursive call visit visits every node in the graph untill it reaches a leaf node so it is $O(n)$. For each vertice the foreachNeighbor() goes over every edge. So every edge and every node is itterated at most. Therefore its runtime is $O(n + m)$.