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 worst case every neighbor of of every neighbor of every neihbor of n could be m vertices or all of them. Therefore its runtime is $O(nm)$.

```
// 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)
           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, size) {
    if(hashNeighbors(N)) {
       foreachNeighbor(N as N1) {
           if(N1.Root.visited == true) {
               size = size + N1.Root.quant
               N1.Root.visited = false // At this point visited should equal
           }
           countRightNodes(N1, size)
       }
    } else {
        return size
    }
}

// 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.

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 $O(n+E+\frac{n-1}{2})$ or $O(n+E)$.