Algorithm Summary Report

Assignment 3

William Corlett

Brandon Forester

Philip Rajala

Robert Millward

The grand design for the program was to implement an Adjacency List structure for use in the assignment parts asking for shortest path, isLoop, and connectedComponents. The final method, msquare, we planned to actually just implement an Adjacency Matrix and perform the matrix multiplication on it to get the 100,100 value of M2. Since the question only asks for cell 100, 100, the Matrices will only consist of 100 by 100 nodes instead of the full 36,000 by 36,000.

For part 1we are asked to find single source shortest path. To reach this objective, we are implementing a DFS method. Here is pseudo-code for that method.

DFS(Graph)

1 for each vertex u that’s an element of Graph.V

2 u.color = WHITE

3 u.pi = NIL

4 time = 0

5 for each vertex u that ‘s an element of Graph.V

6 if u.color == WHITE

7 DFS\_VISIT(Graph, u)

DFS\_VISIT(Graph, u)

1 time = time+1 //white vertex u was just discovered

2 u.d = time

3 u.color = GRAY

//explore edge(u, v)

4 for each v that’s an element of Graph.Adj[u]

5 if v.color == WHITE

6 v.pi = u

7 DFS-VISIT(G, v)

8 u.color = BLACK //blacken u: we are done looking at it

9 time = time + 1

10 u.f = time

The run time of this method along with other DFS methods is O(n+m) as all other DFS methods are.

For part 2, we as a group have determined to interpret the detection of a loop to be checking if there are any self loops. In doing this we are going to use DFS slightly different than our first implementation of it as so:

DFS(Graph)

//this is in our graph function that takes in the result of the DFS at

//the end printing 0 if the result was true and if the result was //false we return to the main function and start DFS with the next

//node to look for more self loops. Once we are done searching through

//all of the nodes, we can finally print 1 for a finding no self loops

8 if result = TRUE

9 print(“0”)

DFS\_VISIT(Graph, u)

4 for each v that’s an element of Graph.Adj[u]

//checking to see if u = v meaning we have found a self loop

//if this is the case, we return true immediately closing the function

//since we only need to find one self loop for the case to be found

5 **if u = v**

6 **return TRUE**

7 if v.color == WHITE

8 v.pi = u

Once again, the run time of DFS is O(n+m).

For part 3 we need to compute the number of connected components in the graph. Since a connected component is essentially a tree within a forest given an undirected graph, all we need to do for this part is once again use DFS and have an integer in main initialized at 0 that we increment after each run of DFS. This is guaranteed to give us the correct number of connected components because in an undirected graph, we will not have to worry about cross edges showing us that a previously thought connected component is actually larger than we predicted.

And again, the run time of DFS is O(n+m).

For part 4 we are making a matrix using the adjacency list. After we make the adjacency matrix, we do the matrix multiplication to compute the entry at 100, 100. We use multiple matrices to prevent memory override from messing up the multiplication

1 for i=0🡪100 //initialize the values of both arrays

2 for j=0🡪100

3 matrix1.val[i][j]=0

4 matrix2.val[i][j]=0

5 matrix3.val[i][j]=0

//update the values of the matrix based on the list values

5 for i=0🡪100

6 for j=0🡪100

//if this is a node we want to update, change to 1

7 if array[i].arrayList.val[j]<100

8 matrix1.val[i][j] = 1 && matrix2.val[i][j] = 1

//skips if the value in the arrayList is greater than 100

9 if array[i].arrayList.val[j]>=100

10 break

//Now that we have our values set we need to do multiplication

11 matrix3[][]= matrix1[][]\*matrix2[][]

The algorithm analysis for this is overall O(n^3). Even if we use a more efficient method to multiply the matrices, as discussed in class, There is not a discovered algorithm that can bring matrix multiplication below O(n^3) .