Henry Daniels-Koch
November 20 2016

Algorithms: Lab 9

1). To find the number of simple paths from s to t, we used a modified form of depth first search that stores the number of paths from each node to t in a table. We first set the count at Table[t] = 1. We then iterate over all of the children of u and first check if the child is the node t. If that is the case, we increase the count for each parent of v by storing their counts in a table. (We might use some sort of linked list to store parents, but that is not shown below). If the node is black, and the node's count is not 0, we increase the count for each parent of v by Table[v]. We also increment the count at the parent u by Table[v] and store this value as Table[u]. If the child is white, then we should explore its children. If the child is black, but does not connect to t, then its count will be 0 and we should go to the next child by not doing anything. Once a node is fully explored, we mark it black. As the end, we return the number of paths at s (Table[s]. This code runs in linear time since it has the same form as DFS which runs in $O(V + E)$ time. The pseudocode is as follows:

```
paths(G,u,t)
    color[u] = grey
    for each v in adjacencyList[u] DO:
        if v == T:
            increase the count for each parent of v by 1 and store in Table
        if color[v] == black and Table[v] != 0
            increase the count for each parent of v by Table[v]
        if color[v] == white
            parent[v]=u
            paths(G,v,t)
    color[u]= black
    return Table[u]
```

2a). To compute the longest path in a DAG starting from a vertex v, we do a DFS on v. We modify the search such that we increment a counter for each new node we travel to. Once we set a node to black, we compare the length of the current path to the max path length so far. This algorithm runs in linear time since DFS runs in linear time and our algorithm is just a modified DFS. The pseudocode is as follows:

```
maxPath = 0
curPath = 0
longestPath(v,curPath)
    color[u] = gray
    curPath ++
    for each v in adj[u]: DO
        if color[v]== white
            parent[v] = u
            longestPath(v,curPath)
    color[u] = black
    if curPath > maxPath: maxPath = curPath
```

b). To compute the longest path in a DAG, we iterate over all nodes and find the longest path from each node. However, we use dynamic programming to store the longest path from each node in a table in order to avoid repeats. Dynamic programming allows our code to run in linear time since any DFS will stop once it hits a node it has already calculated a longest path for. The variables maxPath, and curPath are global. The pseudocode is as follows:

```
maxPath = 0
curPath = 0
```

```
longTotalPath()
    for each node v
        longestPath(v,curPath)
return maxPath
```

I would also need to update longestPath:

```
initialize array T to -1
longestPath(v,curPath)
    if(T[v] ! =-1):
        curPath = T[v]
        if curPath > maxPath: maxPath = curPath
    color[u] = gray
    curPath ++
    for each v in adj[u]: DO
        if color[v]== white
            parent[v] = u
            longestPath(v,curPath)
    color[u] = black
    T[v] = curPath
    if curPath > maxPath: maxPath = curPath
```

c). To compute the shortest paths, we can modify our algorithm from part a. In this version, we store the shortest total path once all of the children have been explored. Similar to part a, this algorithm runs in linear time due to the similarity to DFS which is linear.

```
minPath = V
curPath = 0
shortestPath(v,curPath)
    color[u] = gray
    curPath ++
    for each v in adj[u]: DO
        if color[v]== white
            parent[v] = u
            shortestPath(v,curPath)
    color[u] = black
    if curPath < minPath: minPath = curPath
```

3). To determine whether there is a directed path that visits each vertex exactly once, we calculate the longest path in the DAG using the previous function in 2b. If the longest path is equal to the number of edges, then there must be a directed path that visits each vertex exactly once. This is because the longest path will go through each edge only once. This algorithm runs in linear time since it uses the same code as 2b. The pseudocode is as follows:

```
maxPath = 0
curPath = 0
longTotalPath()
    for each node v
        longestPath(v,curPath)
    return maxPath
```

4). To compute the strong component containing v, we call a DFS or a BFS on the vertex v and store all of the nodes that we hit in an array. We then call a backwards DFS/BFS on the vertex v and store each node that we hit in another array. We then compare the two arrays and return the intersection of the two arrays. This algorithm runs in linear time since DFS and BFS run in linear time and one can find the intersection of the two arrays in linear time as well. The pseudocode is as follows:

```
DFSForward(v)
    color[u] = gray
    array1.push(u)     for each v in adj[u]: DO
        if color[v]== white
            parent[v] = u
            DFSForward(v)
    color[u] = black

DFSBackward(v)
    color[u] = gray
    array2.push(u)     for each v in incoming[u]: DO
        if color[v]== white
            child[v] = u
            DFSBackward(v)
    color[u] = black

SCC()
    DFSForward
    DFSBackward
    print intersection of array1,array2
```

To find all the strong components of a graph, we call the above algorithm SCC on each node. This algorithm runs in quadratic time since SCC is linear and we are calling it on each node. The pseudocode is as follows:

```
allSCCs()
    for each node v: DO
        SCC()
```