CSE 4810: Algorithm Engineering Lab
Lab - 5

Md Farhan Ishmam (180041120)

Group - CSE 1A

April 3, 2023

Task 1

a) The Algorithm

The task is simply finding the number of connected components in a graph. In this case, the vertices are the elements of the array that contains 1. We can have at most 4 edges for the 4 directions. A subgraph will be considered as a component if there are no 1 s to visit in the 4 directions. Trivially, a graph will have no components if all the elements are 0 and will have at least one component if there is at least one element containing 1. We need to count how many such components exist in our given array, and the required algorithm is given below:

- We declare a boolean array called *visited* which has a size equal to our input gird and all the elements of the array initialized to $false$. It means, initially, none of the elements of the graph has been visited.
- We initialize a count variable to 0.
- We loop over our input array by starting from the vertices that contain 1 and hasn't been visited i.e the corresponding value in the *visited* array is $false$. For the elements, we run a graph traversal algorithm which can be either BFS or DFS.
- The graph traversal algorithm will only visit the elements with value 1 and mark them as visited i.e. the corresponding value in the *visited* array will be set to *true*.
- When the queue or stack in the traversal algorithm will be empty, it means there are no other connected elements to be visited from that particular starting node. Then, the count is incremented by 1.
- We will then pick another starting node by continuing to loop over our input array and finding an element that has not been visited and the value is 1. Similarly, we run BFS/DFS from that node. The step is repeated until there are no unvisited elements with the value of 1 remaining in our graph.
- Finally, we return the count variable.

Pseudocode

```
countIslands(grid):
    count = 0
    visited = array(sizeof(grid)) where all the elements are False

    for element in grid:
        if (element == 0) || (visited(element) == True):
            continue          #We skip the element if it is visited or it is 0
        else:
            dfs(grid, element, visited) #dfs takes the graph in the form of the grid matrix
                              #dfs takes the starting node which is the element
            count += 1 # at the end of running DFS the count is incremented by 1

    return count
```

---

b) The Algorithmic Complexity

The complexity of our algorithm is bounded by the complexity of either BFS or DFS. The algorithm can be thought of as running BFS/DFS on all the elements of our input array in the worst-case scenario. Hence, the time complexity is $O(V + E)$ where V denotes the number of vertices or elements in our array and E denotes the number of edges. As an element can have at best 4 edges, the number of edges E is upper bounded by $4V$ i.e. $E \leq 4V$. Hence, the time complexity is $O(V + 4V) = O(5V) = O(V)$. The number of vertices is equal to the number of inputs $m * n$. Hence, the time complexity is simply $O(mn)$.

Task 2

The Algorithm

The modification of the algorithm is given below:

- We first run Dijkstra's Algorithm to find the shortest path. Now, we need to modify the shortest path in such a way that it will become the second shortest path.
- For each edge in the shortest path, we will change the weight to infinity and run Dijkstra's Algorithm again i.e. we are removing one of the edges of the shortest path and finding an alternate path that will be the shortest. We will store the obtained new paths for each edge of our original shortest path as our candidate paths.
- We sort the candidate paths based on the total path cost in ascending order.
- We will return the shortest path from the candidate paths which will have a path cost lesser than our original shortest path (This is done for the multiple shortest path scenario).

Pseudocode

```
calculateCost(path):
    cost = 0
    for edge in path:
        cost += edge.weight
    return cost

secondShortestPath(graph, source, destination):
    shortestPath = dijkstra(graph, source, destination)
    candidatePaths = []

    for edge in shortestPath:
        modGraph = graph.copy()      #Copy our original graph
        modGraph.edge.weight = +inf   #Modify the weight of the particular edge to infinity
        candidatePath = dijkstra(modGraph, source, destination)
        candidatePaths.append(candidatePath)
```

```
candidatePaths = sort(candidatePaths)
leastCost = calculateCost(shortestPath)

for path in candidatePaths:
    pathCost = calculateCost(path)
    if(pathCost < leastCost):
        return path

return null
```

Justification of Correctness of the Solution

If we keep removing an edge and run Dijkstra's algorithm then we will end up with a list of candidate paths that are lesser than or equal to our shortest path. It can be equal because in our graph there might be multiple shortest paths. Our second-shortest path will belong to this list of candidate paths because the second-shortest path will be, at worst, similar to the shortest path but with a slight modification on one of the edges of the path i.e. for a subgraph, the shortest path will not be chosen. That's why we are using a brute force approach by removing each edge of the shortest path and then trying to find the second shortest path.
However, we might have multiple shortest paths. In such scenarios, the candidate path will contain some paths that have a path cost equal to our shortest path. We will simply discard these paths after sorting the candidate paths by comparing the original path cost with the candidate path costs. This ensures that we return such a path that will have more cost than our shortest path but lesser than any other path in the graph. Similar to the shortest path, a graph can have multiple shortest paths and only one of them will be returned.

The Algorithmic Complexity

The time complexity of Dijkstra's algorithm is $O((n+v)log(v))$ where n denotes the number of edges and v denotes the number of vertices in our graph. Dijkstra's algorithm, for every modified edge in our shortest path. Let, the shortest path have k such edge. Then we can rewrite our complexity as $O(k(n+v)log(v))$. In the latter part of the code, sorting is used which has a complexity of $O(klog(k))$, and a for loop is used for comparing every candidate path. The number of candidate paths is equal to the number of edges in the shortest path which has been defined as k. Hence, the overall time complexity is

$$O(k(n+v)log(v) + klog(k) + k) = O(k(n+v)log(v) + klog(k))$$

In the worst case, k can be as large as v i.e. the shortest path will cover all the vertices of the graph. Hence, $k \to v$ and the time complexity will be

$$O(v(n+v)log(v) + vlog(v)) = O(v(n+v)log(v))$$

For a sparsely connected graph, the number of edges and the number of vertices will be roughly equal. Then the time complexity will be

$$O(n(n+n)log(n)) = O(n^2 log(n))$$

The time complexity is significantly higher than the required time complexity of $O(nlog(n))$. I believe, to achieve the required time complexity, Dijkstra's algorithm has to be internally modified. However, if the number of edges is significantly higher than the number of vertices in our graph i.e. $n >> v$, then v can be treated as a constant resulting in

$$\Omega(v(n+v)log(v)) = \Omega(n)$$

Task 3

i) Second Best MST Algorithm

We can either use Kruskal's or Prim's algorithm to find the MST similar to our approach to Task-2. The idea of the algorithm is given below:

- We run either Kruskal's or Prim's algorithm to find the MST.
- For every edge in our original MST, we create a new graph and replace the weight of the edge with a positive infinity value. Then we run Kruskal's or Prim's algorithm again and find a spanning tree that will be stored as a candidate tree.
- We sort the candidate trees based on their path cost.
- We return the least cost tree from the candidate trees that has a cost higher than the MST.

Pseudocode

```
calculateCost(tree):
    cost = 0
    for edge in tree:
        cost += edge.weight
    return cost

secondBestMst(graph):
    mst = kruskal(graph)
    candidateTrees = []

    for edge in mst:
        modGraph = graph.copy()      #Copy our original graph
        modGraph.edge.weight = +inf   #Modify the weight of the particular edge to infinity
        candidateTree = kruskal(modGraph)
        candidateTrees.append(candidateTree)

    candidateTrees = sort(candidateTrees)
    leastCost = calculateCost(mst)

    for tree in candidateTrees:
        treeCost = calculateCost(tree)
        if(treeCost < leastCost):
            return tree

    return null
```

Justification of the Algorithm

The correctness of the algorithm is analogous to the correctness of the modified Dijkstra algorithm used to find the second shortest path. The only dissimilarity is that Dijkstra's algorithm returns a path while this algorithm returns a tree. Calculating the cost of the tree and a path involves the same steps.

The Algorithmic Complexity

The time complexity of Kruskal's algorithm is $O(Elog(V))$ where E denotes the number of edges and V denotes the number of vertices in our graph. The algorithm is run for every edge in the tree. A spanning tree can has $V - 1$ edges as it covers all the vertices and the number of edges in a tree with n vertices is $n-1$. As per previous calculations, the sorting cost is $O(Vlog(V))$ and least cost comparison cost is $O(V)$ Hence, the overall time complexity is:

$$O((V-1)Elog(V) + Vlog(V) + V) = O(VElog(V))$$

Hence, our overall time complexity for the second-best MST is $O(VElog(V))$.

ii) Maximum Spanning Tree

There can be two approaches to finding the maximum spanning tree using Prim's algorithm:

Approach-1 We inverse the weights of the graph and run standard Prim's algorithm. We find the maximum weight w_{max} and we update each weight w_i as

$$w_i = w_{max} - w_i$$

Approach-2 We select the maximum weighted edge instead of the minimum weighted edge in Prim's algorithm. In the end, the algorithm will return the maximum spanning tree.

For both approaches, the time complexity is the same as Prim's algorithm which is $O(Elog(V))$

Task 4

a) The Algorithm

The problem simply requires us to find a directed cycle in our graph. In case a directed cycle exists, the prerequisite conditions will result in a loop and hence, can not be met. The algorithm to find a directed cycle is given below:

1. We use DFS or BFS to traverse the graph represented in the form of a matrix and keep track of the elements in a visited array.
2. If revisit an element that has already been visited, we return False.
3. At the end of the traversal algorithm, we return True.

b) The Algorithmic Complexity

The time complexity is the same as that of DFS or BFS i.e. $O(V+E)$ where V denotes the number of vertices and E denotes the number of edges in our directed graph.