COT 5405 Analysis of Algorithms Fall 2021 Homework #1

UFID: 96703101
Name: *Vyom Pathak*
Instructor: Professor Alin Dobra
Due Date: October 11, 2021

Problem 1

Cycle Finding [50]

Solution

Algorithm

The algorithm to detect and find the cycle in given in 2 parts [Algorithm 1, and Algorithm 2], the first iterates over all the vertices which are not visited, and passes them to a DFS algorithm which makes a DFS tree and tries to find the cycle by checking if it finds a back-edge or not, and continously adds each vertex it has visited till now. If a back-edge is found, it returns TRUE and the algorithm terminates and so does the main algorithm for; and it prints the cycle stored in a global *cycleList*. If the back-edge is not found it returns False. Then, the main algorithm continues to iterate over the vertices to check if any other unvisited vertex can be used to find the cycle until either the cycle is found or the algorithm terminates with a No cycle found message.
By using this scheme, the algorithm is able to find cycle even in disconnected graphs, as if the helper function doesn't find a cycle and still some unvisited vertex is left, those vertices are again used by the main function for cycle finding.

Analysis

Proof of Correctness

The algorithm starts at a vertex v and recursively checks if the last vertex $v^{'}$ has been visited or not. If it is visited, there must be an edge from $v^{'}$ to v. This DFS process runs until we get reach to the vertex from which we started with. As, we are at the starting vertex, we know that there is a path from v to $v^{'}$ where v isn't the parent of $v^{'}$. In conclusion, since there is an edge from $v^{'}$ to v and there exists a path from v to $v^{'}$, a cycle exists in the graph which is found by the given algorithm.

Proof of Termination

If there is a cycle found: The algorithm which uses DFS, starts at a particular start vertex to find a neighbour vertex that has been previously visited before. If the neighbour vertex has been previously visited and the neighbour has a different parent from the start vertex, a cycle has been detected and the algorithm terminates.

If there is no cycle found: The DFS algorithm runs through each vertex atleast once, where each vertex is marked as visited and the number of unvisited vertices decreases. When the number of vertices to be visited is 0 this means that the DFS algorithm has traversed all the vertices in the graph and thus the for loop is terminated which inturn terminates the algorithm.

Running Time

The running time of the algorithm is $O(|E| + |V|)$, when we use the adjacency list representation of the Graph G.
Proof: The DFS algorithm runs for each vertex only once so the out loop in the function has a complexity of $O(|V|)$.
At each vertex, check all it's neighbours i.e. it takes $O(deg(V) + 1)$ time to finish this. Which means that each edge is visited once in total $O(|E|)$ time.

Algorithm 1 Function for finding and printing a cycle in the graph

```
1: Input: Graph G with |V| vertices and |E| edges. Global cycleList[].
2: Output: TRUE if a cycle is found and prints the cycle.
3: function FindCycle(G):
4:     cycle ← FALSE;
5:     for nodes v ∈ V(G) do
6:         if visited[v] = FALSE then
7:             if FindCycleHelper(v, visited[], −1) = TRUE then
8:                 cycle ← TRUE;
9:                 break;
10:            end if
11:        end if
12:    end for
13:    if cycle = TRUE then
14:        root ← cycleList[−1];
15:        for nodes v in cycleList from the end do
16:            println(v);
17:            if v = root && current index in not the last index then
18:                return ; #We have printed the whole cycle.
19:            end if
20:        end for
21:    else
22:        println(No Cycle Found);
23:    end if
24:    return ;
25: end function
```

Algorithm 2 DFS helper function to find the cycle

```
1: Input: visited list visited[], current node v, and parent node parent
2: Output: TRUE if a cycle is found.
3: function FindCycleHelper(visited[], v, parent):
4:     visited[v] ← TRUE;
5:     cycleList.add(v);
6:     for nodes v' ∈ neighbors(v) do
7:         if visited[v] = FALSE then
8:             if FindCycleHelper(v', visited[], v) = TRUE then
9:                 return TRUE;
10:            end if
11:        else if v' ≠ parent then # Check if back-edge is found
12:            cycleList.add(v');
13:            return TRUE;
14:        end if
15:    end for
16:    cycleList.removeLast();
17:    return FALSE;
18: end function
```

After finding the cycle, the algorithm prints the cycle from the *cycleList*, which takes roughly $O(|k|)$ where k is the size of the cycle found.
Even if the algorithm doesn't find the cycle, the algorithm still runs the DFS to check the whole graph and thus it takes $O(|E| + |V|)$. Thus the overall complexity of the algorithm is $O(|E| + |V|)$.

Figure 1 shows the analysis of running time as a function of size of the graph. The evaluation is done for the same in four different ways.

1. By varying the no. of vertices of the graph and keeping the degree of the graph fixed *(Degree=3)*. The *RandomRegularGraphGenerator* [2, 3] constructor was used to generate the graph for the same.
2. By varying no. of vertices and no. of edges as $10 * No.\ of\ vertices$. The *GnmRandomGraphGenerator* [1, 3] constructor was used to generate the graph for the same.
3. By varying no. of edges and vertices as same values i.e. $E = V$. The *GnmRandomGraphGenerator* [1, 3] constructor was used to generate the graph for the same.
4. By creating a Ring graph such that the algorithm has to forceably go around the whole graph inorder to check the worst time complexity of the algorithm i.e. to run all the edges at-least once. The *RingGraphGenerator* [3] was used to generate the graph for the same.

It is clear from the graph that the algorithm's run time follows pseudo linear-time complexity ($O(|E| + |V|)$) for increasing size of the graph.

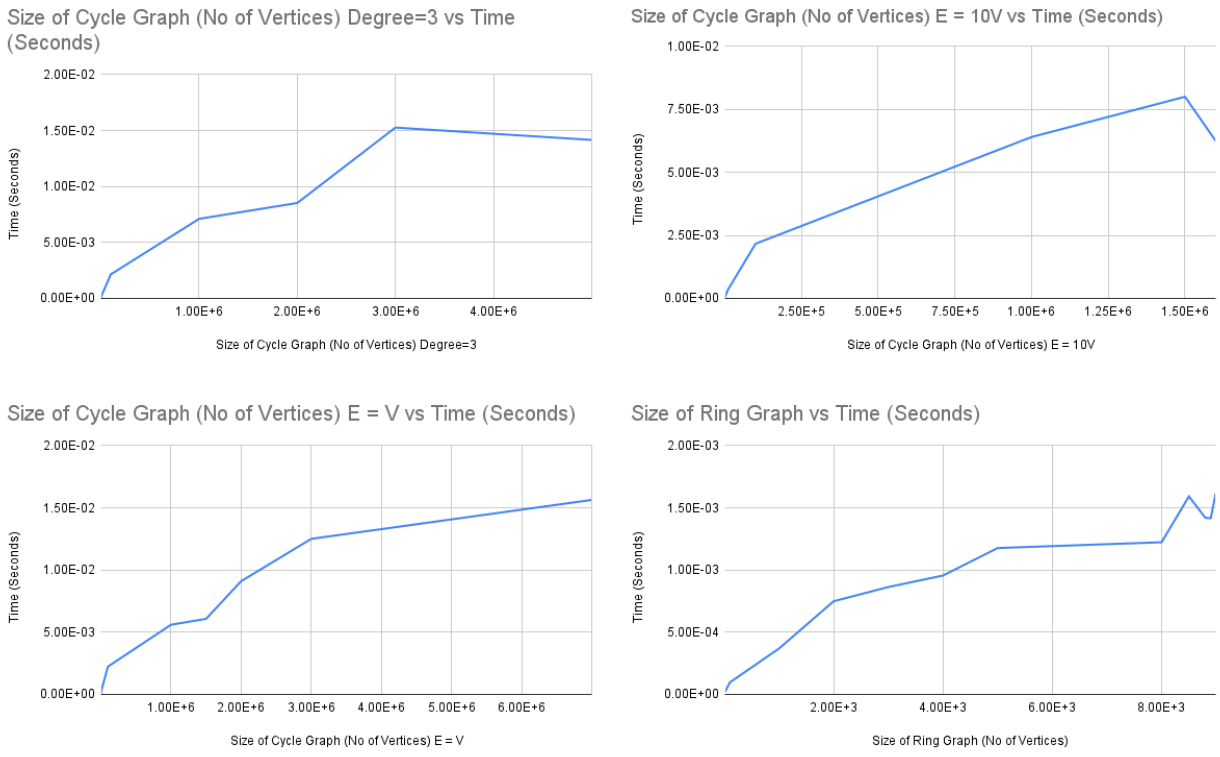

Figure 1: Graphical analysis of running time and complexity of the cycle finding algorithm

Problem 2

Minimum Spanning Tree for sparse graphs [50]

Solution

Algorithm

The algorithm for developing a minimum spanning three for a sparse graph ($|E| = O(|V| + 8)$) using the cycle finding algorithm from the first problem. As there are very few edges, we run the finding algorithm to find a cycle where we remove the max weighted edge. After that we run the find cycle algorithm just that many times such that there are only $|V| - 1$ edges left. The final result left is the minimum spanning tree. The modified algorithm is shown as remove cycle algorithm in Algorithm 3.

This algorithm works because of the red-rule i.e. marking an edge in a cycle with max weight with red-color and discarding it for each $findcycle$ call and then remove all the red edges to get the final minimum spanning tree.

Algorithm 3 Function for removing the max weighted cycle in the graph to form MST

```
1: Input: Graph G with |V| vertices and |E| edges. Global cycleList[].
2: Output: Removes the highest weghted edge from the cycle found.
3: function RemoveCycle(G):
4:     cycle ← FALSE;
5:     for nodes v ∈ V(G) do
6:         if visited[v] = FALSE then
7:             if FindCycleHelper(v, visited[], −1) = TRUE then
8:                 cycle ← TRUE;
9:                 break;
10:            end if
11:        end if
12:    end for
13:    if cycle = TRUE then
14:        root ← cycleList[−1];
15:        for nodes v in cycleList from the end do
16:            Remove the edge e from Graph G, such that W(e) is max of all the edges in the cycle
17:        end for
18:    end if
19:    return ;
20: end function
21: Call RemoveCycle(G) (|E| − |V| + 1) times
22: Print remaining edges gives the minimum spanning three
```

Analysis

Proof of Correctness

The algorithm always finds the MST for Sparse Graphs [$|E| = O(|V| + 8)$]. Assuming that the Graph has at most $|V| + 8$ edges and at least $|V|$ edges.

Proof: This can be proved using a modified red-rule. The red-rule suggests that: Let C be a cycle with no red edges. Select an uncolored edge of C of max weight and color it red [4]. Now, we apply the red-rule after finding a cycle and discard that edge and run the cycle finding algorithm again and apply the red-rule again till we color $|E| - |V| + 1$ edges [$\because$ We want $|V| - 1$ edges to be uncolored for the Graph to form a MST and we have $|E| = O(|V| + 8)$ edges]. Now, the remaining edges form the Minimum Spanning Tree. Essentially, in each cycle finding, we remove an edge i.e. remove a cycle and thus we remove enough edges to

form a tree from the given Sparse Graph. We color the edge with a constraint such that every time we color the max-weight edge. By doing this we ensure that we remove all the max-weighted edges from the graph while keeping the graph connected and remove all the unwanted cycle in the process. Thus, the algorithm always finds the MST. This works for Sparse Graph constraints given in the question because there are at most $|E| - |V| + 1$ cycles, and thus we have to apply this modified rule only constant amount of times.

Running Time
The running time of the proposed algorithm is $O(|E|)$, when we use. the adjacency list representation of the Sparse Graph G.
Proof: The remove cycle algorithm essentially finds cycle and removes the max weight edge using DFS; and thus it takes $O(|E| + |V|)$ time to run as discussed in section 1. Now, we run this algorithm a maximum of $O(|E| - |V| + 1)$ times to remove enough max-weight edges to form the minimum spanning tree. In our case, $|E| = O(|V| + 8)$, thus the remove cycle algorithm runs a max of $O(|V| - |V| + 8 + 1) = O(9)$ times, and each time the cycle finding algorithm runs for $O(|E| + |V|) = O(2 * |V| + 8)$. Thus, the final runtime of the algorithm is $O(9 * (2 * |V| + 8)) \approx O(|V|)$.
Figure 2 shows the analysis of running time as a function of size of the graph. The evaluation is done for the same in the following way.

1. By varying the no. of edges of the graph in the range $|V|, |V| + 8$ and assigning the weights randomly to the edges between the range $100, 500$. The *LinearGraphGenerator* [2, 3] constructor was used to generate a graph with $|V| - 1$ edges and the rest of the edges were added randomly to the graph to create the final graph.

It is clear from the graph that the algorithm's run time follows linear-time complexity ($O(|V|)$) for increasing size of the graph.

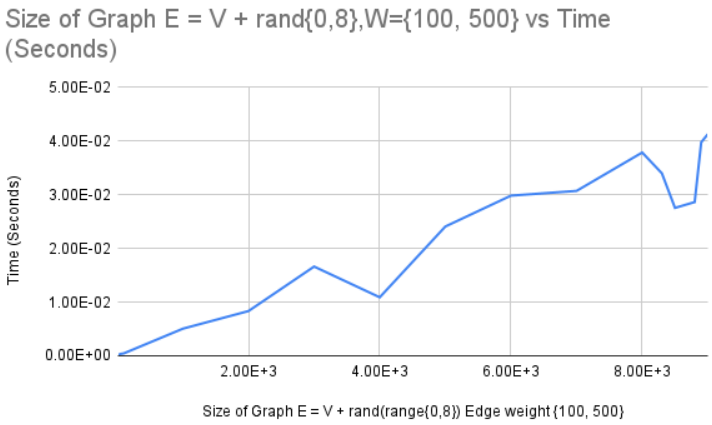

Figure 2: Graphical analysis of running time and complexity of the minimum spanning tree algorithm

Guide to Code

Both the codes were written in Java language.

Steps

1. Download and Install JDK 11+ on your system using the following link: JDK
2. Create new java projects one-by-one and use the $findcycle.java$ and $mst.java$ for problem 1 and problem 2.
3. Download and Add the JgraphT dependency to the java project using the following link: JGraphT.
4. Run the java projects and interactively give input according to the prompt.

* *Tip: Use VS code and Java extensions for easy install and execution.*

The testing of both the codes was done using unit-test cases. Evaluation of the code in-terms of time was done using the $nanoTime()$ *package* of java.

References

[1] Paul Erdös and Alfréd Rényi. On the evolution of random graphs. In *The structure and dynamics of networks*, pages 38–82. Princeton University Press, 2011.

[2] Jeong Han Kim and Van H Vu. Generating random regular graphs. In *Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*, pages 213–222, 2003.

[3] Dimitrios Michail, Joris Kinable, Barak Naveh, and John V. Sichi. Jgrapht—a java library for graph data structures and algorithms. *ACM Trans. Math. Softw.*, 46(2), May 2020.

[4] Robert Endre Tarjan. *Data structures and network algorithms*. SIAM, 1983.