Analysis of Algorithms: Assignment 1

Kaiviswanathan Srikant Iyer (5222-2519)

11th October 2021

1 Problem 1

As part of this problem, you have to design and implement an algorithm to find a cycle (just one cycle) in an undirected graph. Specific tasks you have to accomplish are:

1. Design a correct algorithm and show it in pseudo-code [10]
2. Provide proof of the algorithm's correctness [10]
3. Find and prove the algorithm's running time [10]
4. Implement the algorithm in a compiled language and: [20]
5. Write a graph generator (Hint: use an existing graph generation library if you can)
6. Write test code to validate that the algorithm finds cycles
7. Test the algorithm for increasing graph sizes.
8. Plot the running time as a function of size to verify that the asymptotic complexity in step 3 matches experiments

1.1 Algorithm

Our algorithm makes use of the *jgrapht* [?] library for the purpose of random graph generation. This generation takes a user input - the number of vertices required. We calculate the number of edges in the graph using the equation shown in (1). We make use of the depth first algorithm in order to detect cycles in the randomly generated graph. The depth first traversal algorithm visits each vertex and adds it to the stack. It works on the basis of a recursive function. After visiting a vertex, it visits its adjacent vertex. Then recursively, it's adjacent vertex and so on. The algorithm detects a cycle when it encounters a back edge. This essentially shows us that we are back to where we started. A back edge is detected by checking if we have already visited our current node. (it is already in the recursion stack). This algorithm keeps track of the nodes that we have already visiting using mark(v).

A finite number of test cases were written and tested against the algorithm to check its correctness and complexity. These test cases included corner cases, wrong inputs and complex cases.

$$V = E * (E - 1)/2 \tag{1}$$

1.2 Algorithm Correctness

To prove the correctness of the algorithm, proof by contradiction and proof by termination is explained in this section.

1.2.1 Proof by Termination

Observation 1: All the nodes are visited at least once.
Observation 2: At each vertex, all its neighbours (all its incident edges) are checked.
Claim: Algorithm terminates after after at most V iterations.
Proof: Each time through the while loop an element is checked if it is visited or not, there could be at most V iterations are there are only V vertices in the graph.

1.2.2 Proof by Correctness

Proof: A node is added to the stack after we visit it. The various vertices in the stack form a path. When we revisit a node that is already in our stack, we are essentially back where we started. Thus, we have found a cycle.

1.3 Algorithm Running Time

In this section we analyze the running time complexity of the algorithm to detect a cycle in an undirected graph. We are analyzing the run time complexity of the DFS algorithm that helps us detect a cycle in an undirected graph.

The graph is given in terms of adjacency list because of which:

- The algorithm visits every vertex once. Visiting each vertex takes a constant amount of time. Since the graph has 'V' number of vertices, the run time complexity for visiting each vertex once is $O|V|$.
- When we visit each vertex, we begin exploring the adjacent nodes recursively. Therefore we are traversing the path between the current vertex to the adjacent vertices. Since it takes constant time to traverse an edge, thus the run-time complexity will be $O|E|$.

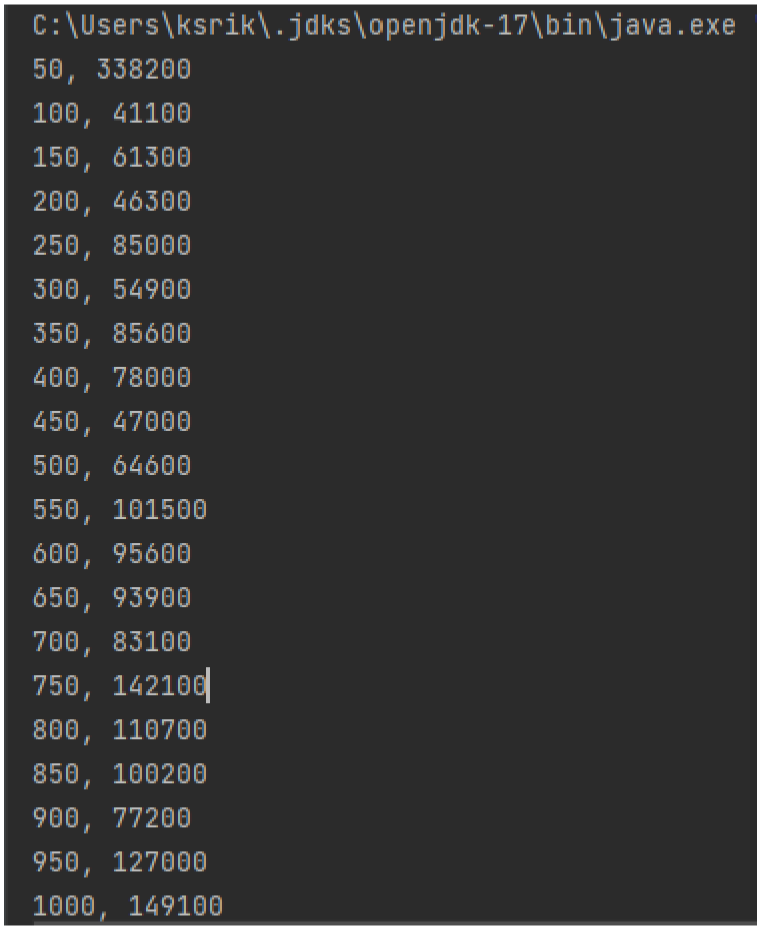
The overall run-time complexity would be $O|V + E|$. As the graph has more vertices or edges, the run time complexity of the algorithm will correspondingly increase.

Algorithm 1: Cycle Finder

```
Input: Number of Nodes to be tested $N$;
Output: Cycle $\mathbb{C}$, if Present;
visit[V] ← False
for 0 → N do
    Nodes are added to the graph G
end
for i ← 0 to V do
    visit(i) = 0
end
for i ← 0 to I do
    move(i) = 0
end
for i ← 0 to V do
    if V ==0 then
        Function calling Deapth First Search(V)
    end
 end
 ————————————————
 Create function Deapth First Search(V) {
 Set(V)=1
 If visited(V)=1
 for adj(V) ← 0 to w do
     if Visited(w)==0 then
         Function Call Deapth First Search(w)
     end
     else
         if Mark(w)=1 then
             Print "Found a cycle"
         end
 end
 }
```

1.3.1 Proof of running time

Proof: In the case where there is no cycle in the graph, we will have to traverse all the edges and vertices in the graph. This is one of the worst cases possible. The other case to consider is when a



```
C:\Users\ksrik\.jdks\openjdk-17\bin\java.exe
50, 338200
100, 41100
150, 61300
200, 46300
250, 85000
300, 54900
350, 85600
400, 78000
450, 47000
500, 64600
550, 101500
600, 95600
650, 93900
700, 83100
750, 142100
800, 110700
850, 100200
900, 77200
950, 127000
1000, 149100
```

Figure 1: Result of Minimum Spanning Tree.

cycle is present in the graph that includes all the vertices of the graph. In this case as well, we will have to visit all the vertices and edges of the graph.

1.4 Results

The figure shows the relation between the number of nodes and the running time (in milliseconds) of the Depth First Search algorithm. The graph should resemble a O(ElogV) curve however since the graph generated are random and it doesn't produce the exact expected output curve.

2 Problem 2

As part of this problem, an efficient algorithm was designed and implemented to find the minimum spanning tree.

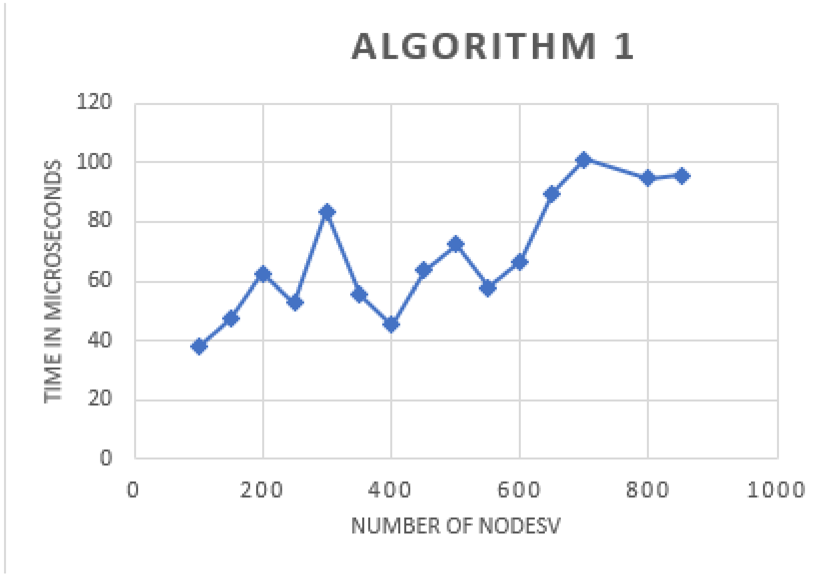


Figure 2: Asymptotic Complexity Cycle Finder Algorithm.

2.1 MST Algorithm

The problem has been solved by designing an algorithm to find the minimum spanning tree in an undirected sparsely connected graph. A spanning tree is one that connects all the edges of a tree, with the minimum number of edges such that there is no cycle. A minimum spanning tree is one that has edges with the minimum sum weights of their edges. Considering a graph with V vertices the spanning tree has $V - 1$ edges.

The algorithm takes a random graph as input. The random graph is generated by connecting random nodes. The number of vertices V is taken as input from the user and the weights and number of edges are generated randomly. The number of edges is between N and $N+8$ where N is the number of vertices.

In order to construct a minimum spanning tree, the Kruskals algorithm is used. Initially the algorithm begins by sorting the edges according to their weight. The edge with the minimum weight is chosen, and it is checked whether this edge results in a cycle in the partially constructed spanning tree. If it results in a cycle, the edge is discarded. Else, it is added to the minimum spanning tree. This process is continued till we obtain $V - 1$ edges.

For the purpose of cycle detection we make use of the Union find algorithm within the Kruskal's algorithm. Appropriate test cases were written and the implementation has been checked for its correctness and run time complexity.

2.2 Algorithm Correctness

To prove the correctness of the algorithm, proof by contradiction and proof by termination is explained in this section.

Algorithm 2: Minimum Spanning Tree in a Undirected Graph

```
Input: A Undirected Graph $G$;
Output: MST $\mathbb{M}$, if Present;
MakeNewSet(x)
. Place element $x$ in its own set
Compare(x,y)
. Return true if $x$ and $y$ are in the same set
Join(x,y)
. Combine sets containing $x$ and $y$
Let V = Null
Let A be a disjoint-set data structure
for each x belongs to X do
    Call A.MakeNewSet(x)
end
for each edge (x, y) sorted by cost do
    if A.Compare(x, y) is false then
        Add (x, y) to V.
        Call A.Join(x, y).
    end
end
print MST graph
```

2.2.1 Proof by Contradiction

Claim: T is the minimum spanning tree obtained as a result of Kruskal's algorithm and T' be the hypothetical spanning tree such that the weight of the hypothetical tree is lesser than the weight of the minimum spanning tree obtained by Kruskal's algorithm.

Proof: As T' is the hypothetical minimum spanning tree, then $W(T')<W(T)$. This leads to the understanding that since the sum weights of the trees are different they must have different edges.

Let us assume that the following edge difference exists between spanning trees T and T', T has an edge e whereas T' has an edge e'. Since the Kruskal's algorithm chooses the minimum edge at each step, considering it has chose the edge e over e', it is understood that $W(e')<W(e)$.

Reducing the edge difference between T and T', the edge e' in T' can be replaced by edge e. This means that the sum of the weights of the spanning tree will reduce, but this is contradicting with the fact that T' is the minimum spanning tree.

2.2.2 Proof by Termination

Claim: The algorithm iterates to at most $V + 8$ or E iterations.
Proof: Kruskal's algorithm finds the minimum edge that doesn't result in cycle formation, to add to the spanning tree. A minimum spanning tree has $V - 1$ edges where the original graph has V vertices. The worst case results in traversing all the edges in the graph. Thus, according to the given problem specification, the number of edges in the random graph will be at most $V + 8$. Thus, the worst case of the algorithm results in $V + 8$ iterations.

2.3 Algorithm Running Time

This section explains the runtime analysis to find a minimum spanning tree using Kruskal's algorithm. Kruskal's algorithm begins by sorting the edges according to weight. This takes O(ElogE) time. After sorting the edges, the algorithm proceeds to perform cycle detection. The union find algorithm is used to ensure there are no cycles in the constructed spanning tree. This operation takes O(logV) time. Thus, the overall time complexity of the algorithm is O(ElogE+ElogV) or O(ElogV).

2.4 Results

The figure shows the relation between the number of nodes and the running time of Kruskal's algorithm in milliseconds. The graph resembles the running time of Kruskal's algorithm, $O|ElogV|$. The graph generated is random and thus, it doesn't produce the exact expected output curve.

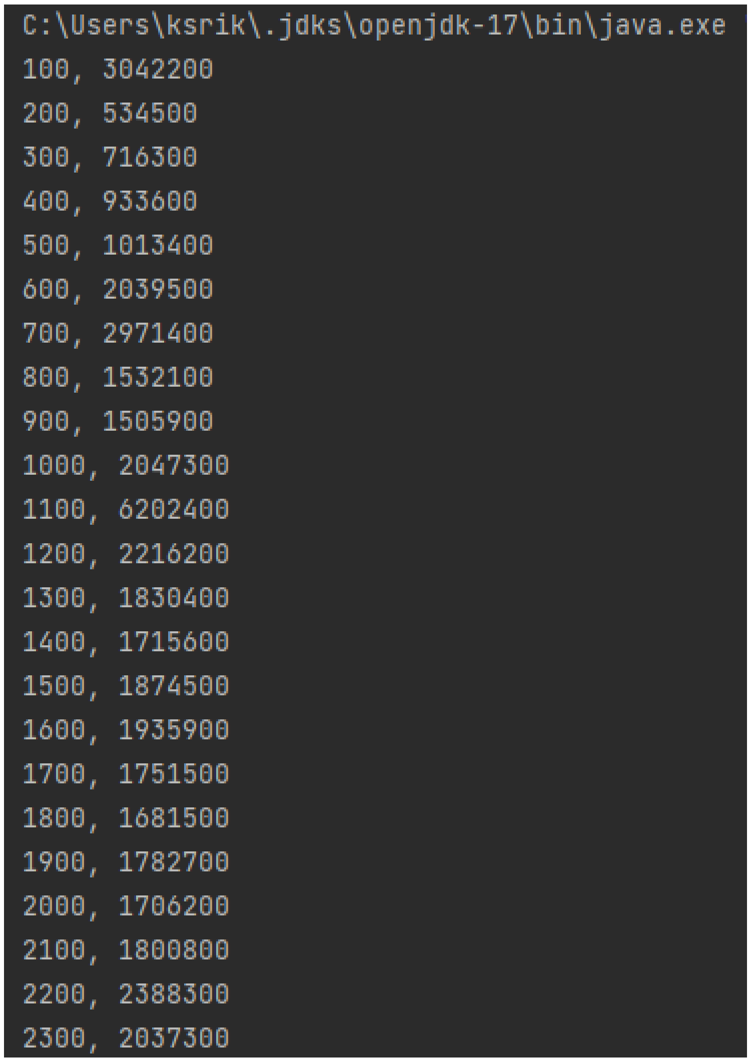
3 Running the Code, Testing and Evaluation

To run the code.

- Install java
- get Jgrapht
- Run "Question1Cycle.java"
- Run "Question2MST.java"

The following tests were dont on the code to verify its correctness:

- For question 1 the cycles were tested for test cases 0,1 and 2 where result came as "Cycle not found".
- For 3 Nodes, cycle's presence was not found in some cases.
- For more nodes as the graph was randomly generated the number of cycles were harder to find in graphs with less number of edges and low number of vertices.



```
C:\Users\ksrik\.jdks\openjdk-17\bin\java.exe
100, 3042200
200, 534500
300, 716300
400, 933600
500, 1013400
600, 2039500
700, 2971400
800, 1532100
900, 1505900
1000, 2047300
1100, 6202400
1200, 2216200
1300, 1830400
1400, 1715600
1500, 1874500
1600, 1935900
1700, 1751500
1800, 1681500
1900, 1782700
2000, 1706200
2100, 1800800
2200, 2388300
2300, 2037300
```

Figure 3: Result of Weighted graph.

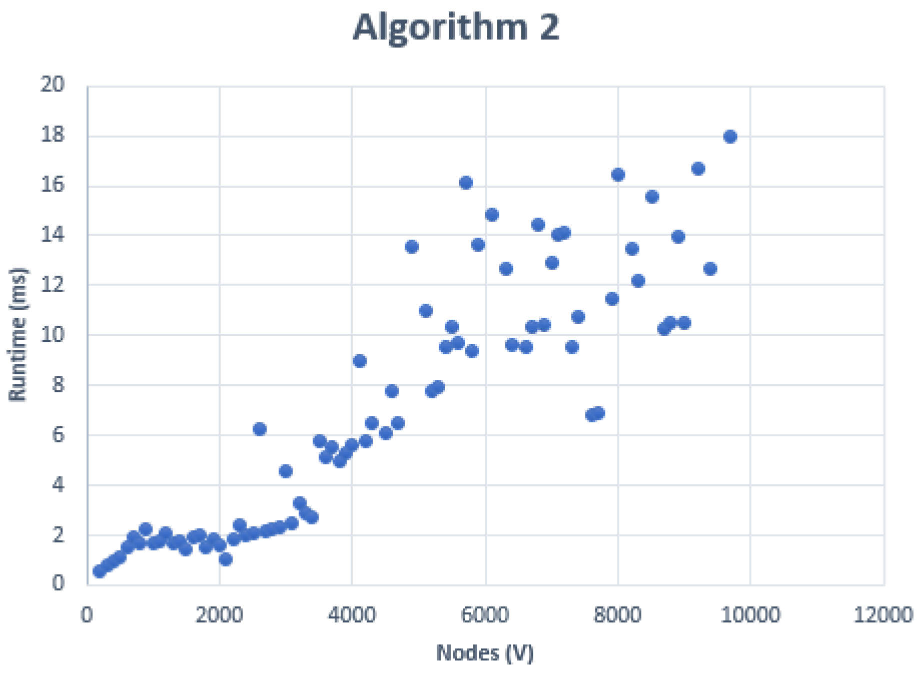


Figure 4: Asymptotic Complexity of MST.

- In Question 2 the we tested the output by printing out the values and plotting the weighted graph.
- Such testing was not possible for larger graphs for which the testing was done by providing increasing weights and calculating N*(N-1)/2

4 Observations and Assumptions Made

Following assumptions have been made to reduce ambiguity of the problems given -

- The graphs and edges are generated randomly. Thus the output curves of the graphs do not follow a gradual curve as it should for both the problems.
- An assumption has been made for the relation between the number of nodes and number of edges for problem 1.
- The Kruskal's algorithm makes use of the union find algorithm for cycle detection. Thus, it is assumed that there are no self loops in the graph.