COT5405 - Analysis Of Algorithm

Assignment 1 - Greedy Algorithms

Submitted By: Karan Asthana

Computer and Information Science and Engineering Department
Fall 2021
University of Florida
October 11, 2021

1 Cycle Finding in an Undirected Graph

1.1 Algorithm Pseudo Code

The basic idea of the algorithm is to traverse the undirected Graph in a Depth-First Search (DFS) and then keep a track of the nodes/vertices that we've already visited. As soon as a node appears twice in the visited list, it is because of a cycle present in the graph. That is, there are two different ways of reaching a Node n. We take extra care in the algorithm to pass on the parent of the current node to the next execution of the recursive DFS function (so as to prevent traversing in the backward direction, which would essentially lead to infinite loops or faulty cycle detection).

We start from the initial node and traverse all the edges connected with it recursively. If we, at any time encounter the same node again, we track it and return the cycle according to the traversed nodes in this execution.

The pseudo code for the algorithm is stated on the next page

Algorithm 1 FindCycle(G)

```
function FindCycle(graph)
    vSet ← graph.vertices(), visited ← [], cycle ← [], stack ← newStack()
    for Vertex v in vSet do                          ▷ this is to incorporate non-connected graphs
        if visited.containsv then continue
        end if
        stack.clear()
        if DFS(v, parent) then                       ▷ parent is null or -1 at start
            start ← stack.top
            cycle.append(start)
            while stack ≠ Empty do
                if stack.top=start then
                    break
                end if
                cycle.append(stack.top)
            end while
            cycle.append(start)
            return cycle
        end if
    end for
end function

function DFS(v, p)                  ▷ v is the vertex              ▷ p is the parent of vertex v
    visited.append(v)
    stack.append(v)
    for NeighborVertex n in v.neighbors() do
        if (not visited.contains(n) AND DFS(n, vertex)) then
            return true
        end if
        if n ≠ parent then          ▷ Adding n to the stack, if the previous node was not n
            stack.append(n)
            return true
        end if
    end for
    stack.pop()
    return false
end function
```

1.2 Proof of Correctness

Lemma: If a graph G contains a cycle. Then during graph traversal, we visit the same vertex twice, it means that there is at least one cycle in the graph.

Proof. **Using Contradiction:**

Let us assume that we have a cyclic graph G with vertices $v_1, v_2....v_n$ and edges $e_q, e_2....e_n$.

Let us assume that the graph G has 1 cycle from the vertices v_j to v_k and during traversal v_j is visited only once.

Start traversing the graph starting from the parent vertex of v_j.
Continue traversing till all the nodes have been traversed.

But on encountering the cycle, the edges again lead to the same vertex v_j (after the first traversal).

This is a contradiction, since initially we had assumed that every node is visited only once. But in order for a cycle traversal to complete, we have to visit the same vertex more than once. □

Thus, a graph G that contains at least a cycle, visits at least one node twice.

1.3 Algorithm Running Time

The worst case time complexity is **O(V+E)**, where V is the number of vertices and E is the number of Edges.

Proof. According to the pseudo code mentioned above, in the worst case, we are traversing **all** the vertices once using **all** the edges twice (since in Undirected graphs, every edge can be considered twice during traversal, which is ignored via the parent logic in our algorithm). So, intuitively our time complexity is in the order of the sum of number of vertices and the number of edges

- In the pseudo code, we find all the vertices and start traversals from these nodes (if not visited before) (this is equivalent to a worst-case O(V) time)
- We then traverse all the edges connected to a vertex v, using the Depth First logic. That is, we take one vertex and then using one of its edge, go to the next vertex and then, so on.
  Thereby, we traverse all the edges at max twice, therefore, it is equivalent to O(2E).
- Other operations, such as finding max in a cycle are always of a lower order, usually taking constant time.
- Therefore, the effective worst case complexity of the algorithm leads to O(V + 2E + V + C), which is equivalent to **O(V+E)**

□

1.4 Implementation

1.4.1 Graph Generator

For the implementation of the algorithm, a graph generator library, **jgrapht** (written in Java) was used.
The number of vertices in the order of 10^6 were randomly selected.
The number of edges in the order of $2 * numVertices$ were randomly selected.
The combination of these number of edges and vertices was then input to the graph generator library, which then returned a Graph object with the specified vertices and unweighted edges.

1.4.2 Test code to validate algorithm correctness

For testing the correctness of the algorithm, random small test cases were written whose result was already known. The results of the algorithm were then compared to the already known results.

Attached image Figure 1.1, shows a screenshot of the test cases output as seen on the terminal while execution.

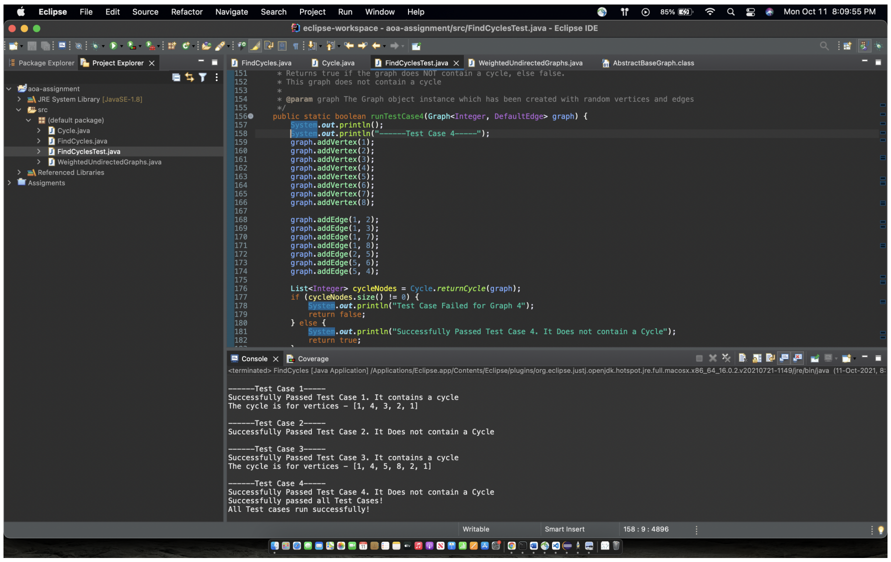

Figure 1.1

Attached Figures 1.2, Figure 1.3, Figure 1.4 and Figure 1.5, shows the test case inputs that were tested by hard-coding their values.

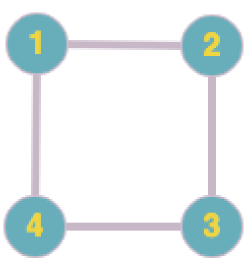

Figure 1.2 (Test Case 1 - Contains Cycle - 1,2,3,4,1)

Figure 1.3 (Test Case 2 - Does not Contain Cycle)

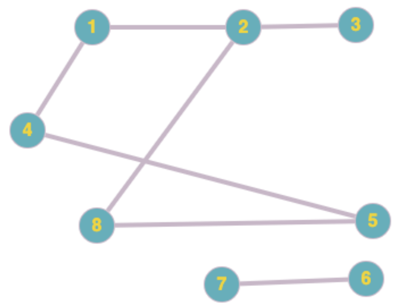

Figure 1.4 (Test Case 3 - Contains Cycle - 1,2,8,5,4,1)

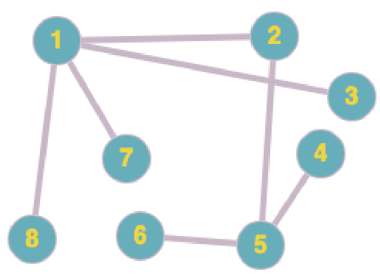

Figure 1.5 (Test Case 4 - Does not Contain Cycle)

1.4.3 Test for increasing graph sizes

Using a for loop, the number of vertices were randomly selected, leveraging the random function for a fraction of 1 million nodes in one execution.

---

for i in $(1, 2000)$ **do**
  $numVertices \leftarrow (random() * 10^6)$
  $numEdges \leftarrow (random() * 2 * numVertices)$
  $jgrapht.generateGraph(numVertices, numEdges)$
  return *graph*
end for

1.4.4 Run time vs number of nodes Plot

Attached image Figure 1.6, shows a screenshot of the run time vs number of nodes plotted on a graph.
The x-axis contains the number of nodes (with respect to 1million)
The y-axis contains the time of execution (in nano seconds, with respect to 1second)

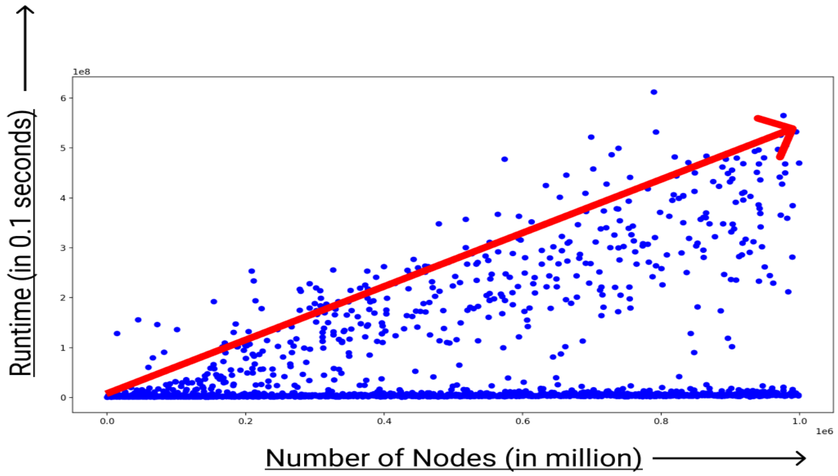
Runtime (in 0.1 seconds)
Number of Nodes (in million)
1e8
1e6
0
1
2
3
4
5
6
0.0
0.2
0.4
0.6
0.8
1.0

Figure 1.6

2 Minimum Spanning Tree for "sparse" graphs

2.1 Algorithm PseudoCode

The basic idea of the algorithm is to leverage a modified version of the cycleFinding algorithm (ref: Algorithm 1), which returns the heaviest in the first cycle that is found in the graph.
This process is repeated until there are no cycles in the weighted graph. (which is also $numCycles \leftarrow (numEdges - numVertices + 1)$)

The pseudo code for the algorithm is stated below

Algorithm 2 FindMST(G)

---

function FINDMST($graph$)
  $vSet \leftarrow graph.vertices()$, $eSet \leftarrow graph.edges()$
  $numVertices \leftarrow vSet.size()$, $numEdges \leftarrow eSet.size()$
  for i in $(numVertices + 1 - numEdges)$ **do** ▷ this is to remove the extra edges
    $cycle \leftarrow FINDCYCLE(graph)$
    $heaviestEdge \leftarrow cycle.heaviestEdge$ ▷ this returns the cycle as well as the heaviest edge in it
    $graph \leftarrow graph.delete(heaviestEdge)$
  end for
  return graph
end function

2.2 Proof of Correctness

Lemma: A Minimum Spanning Tree with n vertices has n-1 edges.

Proof. **Using Mathematical Induction**

Base Case (n=1):
Let us assume that we have a tree with only 1 node and the number of edges in this tree = 0 (i.e. n-1).

Case (n=2):
Let us assume that we have a tree with only 2 nodes and the number of edges in this tree = num(0) + 1(since it has to connect to only 1 existing node so as to not form any cycle)
Thus, n-1 = 1 edges.

Case (n=n):
Let's assume that the inductive step is true, i.e. for n nodes, number of edges = n-1 edges.

Case (n=n+1):
The number of edges will be (n-1) + number of edges required to connect to the n node tree
Thus, the total number of edges will be (n-1) + 1 = n edges.

Thus, we can say that Case(n+1) is true.

Hence, using the Principle of Mathematical Induction, it is proved that for n vertices, a minimum spanning tree will contain (n-1) edges.

In our algorithm's pseudo code, we are reducing the number of edges to n-1 and hence, we can conclude to say that our algorithm is correct.

□

2.3 Algorithm Running Time

The worst case time complexity is **O(V)**, where V is the number of vertices.

Proof. According to the given problem statement, in the worst case, we have (V+8) edges, i.e. 9 cycles in the graph. So, to form an MST, we need to obsolete out all the cycles in the graph (by removing their heaviest edge). By (1.3), we know that finding a Cycle takes **O(V+E)** worst-case time. Since, we have a maximum of 9 cycles, we will take
$9 * (find\_heaviest\_edge\_in\_a\_cycle) * (removal\_of\_heaviest\_edge)$
time to remove all the cycles and thus, form the MST.

- Using a modified version of Algorithm 1, getting the heaviest edge of a cycle from the graph takes O(V+E) time, where E = (V+9), i.e. O(V+V+9), i.e. **O(2V+9)**
- Doing the above operation for all the extra edges (9), we get the overall run-time as 9*O(2V+9), i.r. **O(18V + 81)**, which is effectively O(V).

□

2.4 Implementation

2.4.1 Graph Generator

For the implementation of the algorithm, a graph generator library, **jgrapht** (written in Java) was used.
A weighted Tree was initially generated for a random number of vertices (it will have exactly 1 lesser edge than the number of vertices)
After generating a weighted tree, it was converted into a graph object and then, random number of edges (between 0 and 9) edges were added to the tree with random weights.
The number of vertices in the order of 10^6 were randomly selected.
The number of edges (between $numVertices - 1$ and $numVertices + 8$) was randomly selected.

2.4.2 Test code to validate algorithm correctness

For testing the correctness of the algorithm, random small test cases were written whose result was already known. The results of the algorithm were then compared to the already known results.

For testing the MST, we have used the resultant edges of the MST as a way of testing correctness. (If the edges in the resultant MST are correct, it was an MST)
Attached image Figure 2.1, shows a screenshot of the test cases output as seen on the terminal while execution.

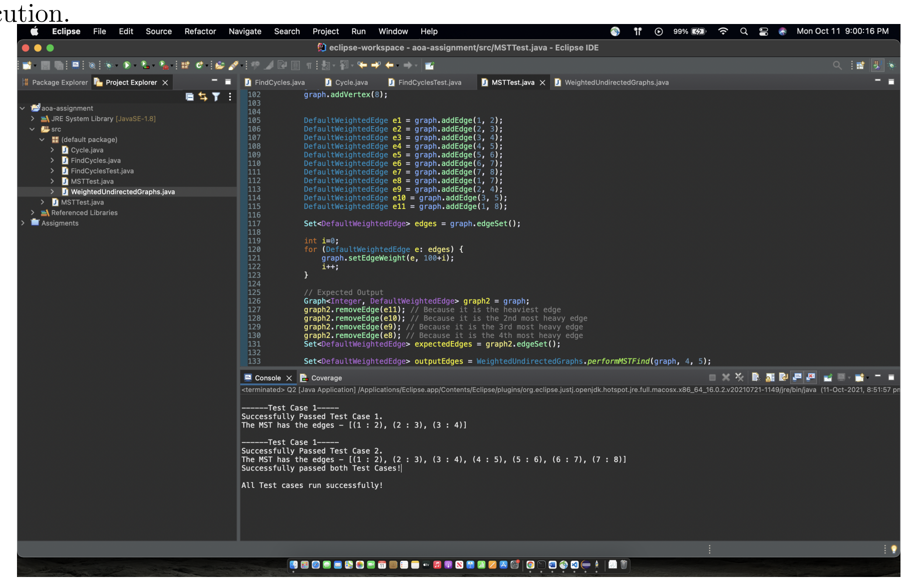

Figure 2.1

Attached Figures 2.2 and Figure 2.3, shows the test case inputs that were tested by hard-coding their values.

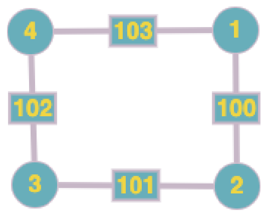

Figure 2.2 (Test Case 1 - Contains MST with edges - [(1 : 2), (2 : 3), (3 : 4)])

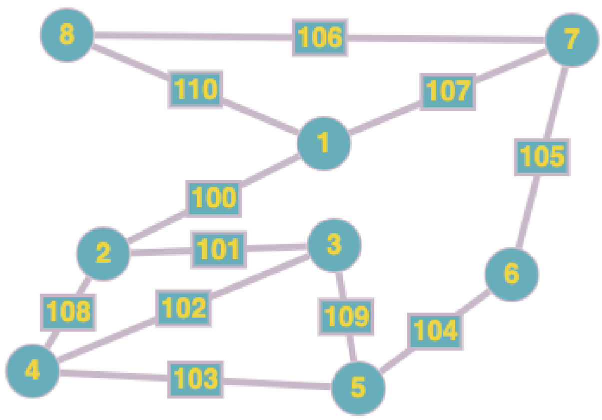

Figure 2.3 (Test Case 2 - Contains MST with edges - [(1 : 2), (2 : 3), (3 : 4), (4 : 5), (5 : 6), (6 : 7), (7 : 8)])

2.4.3 Test for increasing graph sizes

Using a for loop, the number of vertices were randomly selected, leveraging the random function for a fraction of 1 million nodes in one execution.

for i in $(1, 2000)$ **do**
    $numVertices \leftarrow (random() * 10^6)$
    $extraEdges \leftarrow (random() * 9)$
    $graph \leftarrow jgrapht.generateWeightedTree(numVertices)$
    for j in $extraEdges$ **do**
        $graph$.addEdge($randomWeight$)
    end for
    return $graph$
end for

2.4.4 Run time vs number of nodes Plot

Attached image Figure 2.6, shows a screenshot of the run time vs number of nodes plotted on a graph.
The x-axis contains the number of nodes (with respect to 1million)

The y-axis contains the time of execution (in nano seconds, with respect to 1second)

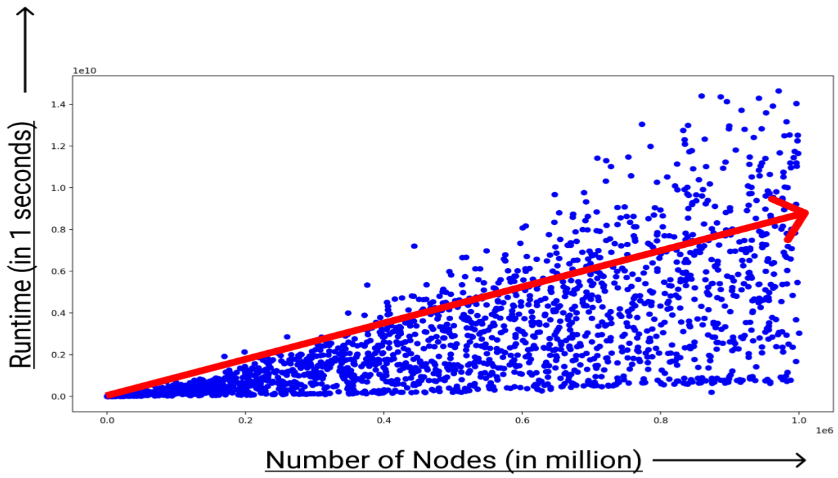

Figure 2.6 - time VS num(nodes) plot

3 Solution Specifics

3.1 Installation and Setup

Installations

- Install Java SDK 1.8
- Download jar file for the Java Library, jgrapht
- Download an IDE, Eclipse for running the code
- Add the jar file for jgrapht-core in the eclipse project
- Install Python (for graph plotting of run-time vs number of nodes)

Assignment Question 1

- For running the Assignment Question 1, create a Run configuration that runs the code starting from the Java file, FindCycles.java
- This file internally runs a loop for a large number of times (currently set to 100) and finds out a cycle in a randomly generated graph.
- If there is a cycle present in the graph, the vertices of the graph are printed on the console output.
- If there is no cycle present in the graph, a message is printed for the same.

Assignment Question 2

- For running the Assignment Question 2, create a Run configuration that runs the code starting from the Java file, "FindMST.java" (earlier named as WeightedUndirectedGraph.java)
- This file internally runs a loop for a large number of times (currently set to 100) and finds out the MST in a randomly generated graph.
- The randomly generated graph is first created by creating a random weighted tree and then manually adding up to 9 more weighted edges.
- The corresponding edges contained in the MST are then printed on the console output.

3.2 Testing

- Separate files named FindCyclesTest.java and MSTTest.java have been created for specifically testing the correctness of the algorithms.
- For testing of the first question of the Assignment, the initial block of lines in the main functions of the **"FindCycles.java"** and the **"FindMST.java"** files have to be uncommented presently.

- The file **"FindCyclesTest.java"** contained 4 hard-coded different test cases, 2 of which had no cycles in the graph. While 2 of them contained cycles.
- The runTestCases function was run, which returned a boolean value for passing or failure of test cases. Along with relevant console messages.
- These test cases have been explained above with images.

- The file **"MSTTest.java"** contained 2 hard-coded different test cases.

- Test Case 1 contained a graph with 4 nodes and 5 edges with varying weights. The output returned was compared to be the exact 3 edges which had the least weights and formed a connected graph.
- Test Case 2 contained a graph with 8 nodes and 11 edges with varying weights. The output returned was compared to be the exact 7 edges which had the least weights and formed a connected graph.
- The runTestCases function was run, which returned a boolean value for passing or failure of test cases. Along with relevant console messages.
- These test cases have been explained above with images.

Graph Plot of run-time

- To map the run time as a function of the number of nodes, the run time for every execution was noted and a CSV was created which mapped the number of nodes, number of edges and their run-time (in nanoseconds).
- Using Python, and the matplotlib library, a graph was generated between the number of nodes and the runtime of each execution.