UFID : 8296-7553 COT 5405
Saksham Bathla **Homework 1** October 12, 2021

Problem 1 : Cycle Detection

Given an undirected graph G (V,E), we have to identify a cycle in the graph.
Algorithm:

Algorithm 1 Cycle Detection Using DFS

```
1:  procedure DetectCycle(graph)
2:      for vertex in graph.vertexSet do
3:          if visited[vertex] $\leftarrow$ true then
4:              continue
5:          if $DetectCycleUtil(vertex, null, visited, graph)$ then return true
        return false
6:  procedure DetectCycleUtil(source, parent, visited, graph)
7:      visited[vertex] $\leftarrow$ true
8:      $adjacentVertices \leftarrow graph.adjacencyList[vertex]$
9:      for adjVertex in adjacentVertices do
10:         if visited[adjVertex] $\leftarrow$ true then return true
11:         if $DetectCycleUtil(adjVertex, vertex, visited, graph)$ then return true
        return false
```

1. **Proof of Correctness:**

   G = (V,E)
   Let us assume that there exists a cycle in the graph after the completion of the algorithm having not found any cycle.
   $\implies$ $\exists$ *a path P in G s.t. P starts with a vertex v and ends with the same vertex*
   But as we use recursion to traverse all possible paths starting from v and stop on reaching v again
   $\therefore$ *the path P would have been found in the running of the algorithm*
   $\implies$ *Our assumption was incorrect*
   Hence proved

2. **Proof of Runtime:** If there exists a cycle, then worst case will be that all nodes are a part of the cycle. In this case, the DetectCycle procedure would visit each vertex once and each edge only twice since we are keeping a track of all nodes visited previously. Hence, the complexity would be O(V+E).
   In case of no cycle, then recursion would still visit each vertex once. Hence, the complexity would still be O(V+E).

Figure 1 and **Figure 2** show the plot of time taken in case of worst case as well as random inputs.

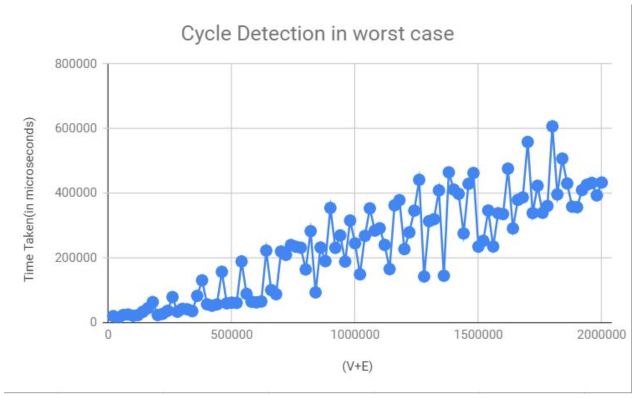

Figure 1: running time as a function of size in case of worst case input

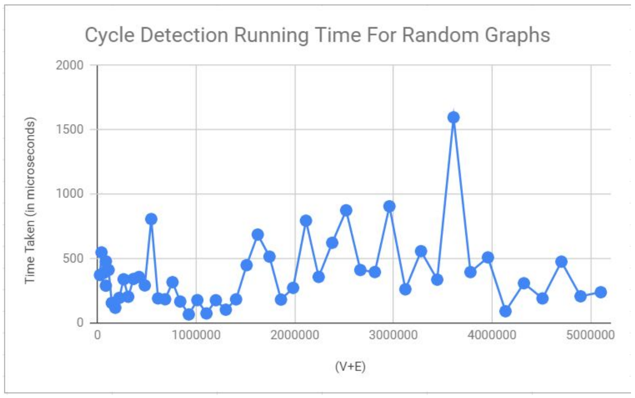

Figure 2: running time as a function of size in case of random input

Problem 2 : Minimum Spanning Tree for Sparse Graphs

For a sparse graph G(V,E), with maximum number of edges = number of vertices + 8, we have to find the MST of the graph

Algorithm 2 MST using Reverse Delete Algorithm

```
1: procedure GETMST(GRAPH)
2:     $edgeSet \leftarrow sort(graph.edgeSet)$
3:     $mstWeight \leftarrow 0$
4:     $i \leftarrow numEdges - 1$
5:     while $i \geq 0$ do
6:         $edgeSet' \leftarrow removeEdge(edgeSet[i])$.
7:         if $isConnected(G, edgeSet') \neq true$ then
8:             $mstWeight \leftarrow WeightOf(edgeSet[i])$
9:             $edgeSet' \leftarrow addEdge(edgeSet[i])$.
10:        $i \leftarrow i - 1$.
       return $mstWeight$
```

Proof of Correctness :

Proof That the final graph is a spanning tree: We reverse sort the edges of the connected Graph G(V,E) by weight and start removing edges from G such that G'(V,E') = G(V,E) – max(e). Then we check using isConnected(), if the graph remains connected. If the graph gets disconnected, we add the edge back. Hence, after each operation the graph results in a connected graph. We keep repeating this operation until no cycles are found. Hence, this results in a spanning tree.

2. **Proof of Minimum Weight:**

   We begin with sorting the edges such that w$_{e1} > w_{e2} > w_{e3}\ldots\ldots > w_n$

During each iteration we remove the maximum weight edge if it doesn't lead to disconnection.

If we assume that the resulting graph is not the minimum spanning tree, this implies that there must be an edge in the final graph with higher weight then one of the removed edges which could have been removed.

But, since we started with removing higher weights edge and checked the connectivity at each iteration, It means that the higher weight edge must have been removed first if it didn't lead to a disconnected graph.

Hence, no edge exists in the graph that could be replaced with a lesser weight edge resulting in an MST.

Hence proved.

Proof of Runtime: O(E.log(E))

In this implementation, we first sort the edgeSet, which happens in O(E.log(E)) time. Then, for each edge, we perform the find and union in our disjoint set of vertices (with union by rank and path compression) which gives a complexity of O(V) Since V E, our Complexity becomes O(E.log(E)).

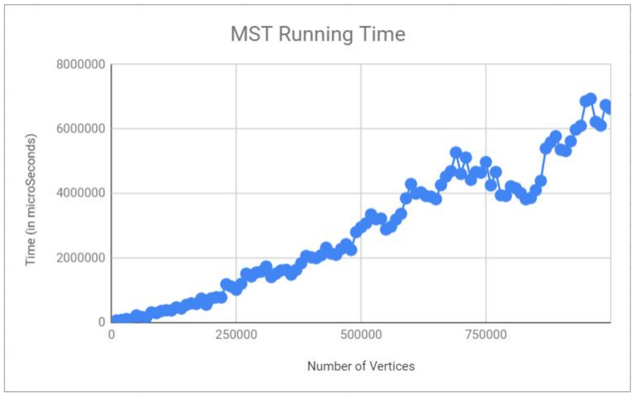

Figure 3: running time as a function of size in case of random input

Implementation

1. **Running this code:**

   Problem 1 :
   The graph generator generates a graph using Jgrapht library graph generation function. I have also generated a worst case graph generation function where we create a random tree with no cycles, so as to test the algorithm for the worst case inputs

   Code File: CycleFinder.java

   To run: java -cp lib/jgrapht-core-1.5.1.jar CycleFinder.java

   Problem 2: Here we start by generating a random tree by making randomly selecting children for each node and avoiding any back edges. After doing so, we add a random number of edges (between 0 and 9) in the graph to simulate the sparse graph

   Code file : Mst.Java

   To run: java -cp lib/jgrapht-core-1.5.1.jar Mst.java

2. **Testing and evaluation:**

   Problem 1:

   1). I generated different acyclic graph of various sizes too test the correctness and worst case runtime of the algorithm. I checked the correctness of the algorithm against the jgrapht library functions available to find cycles and to check for acyclicity

   2). I generated various random graphs using the Jgrapht random graph generator and tested the time taken as well as correctness against the library functions available in jgrapht. For testing correctness, I also generated some small sized hard-coded graphs to check the corretness of the algorithm

Problem 2:

1). Here we started by generating a random tree by making randomly selecting children for each node and avoiding any back edges. then I added random edges to this tree to produce the required sparse graph. To test for correctness, I checked it against the library function for MST in jgrapht and ran it on vertex sizes upto 1000000 to plot the time complexity.