**Report – Course Project**

**CSCE 629-601 Analysis of Algorithms**

**Name: Stone Chen**

**Date: 11/30/2021**

**The overview of the implementation:**

The network optimization is implemented in Java. It contains 6 classes in the package, including “Edge”, “Graph”, “GraphGenerator”, “MaxHeap”, “MaxSortingHeap” and “main”. The first 5 classes are parts of the implementation while the last one, “main”, is used for testing the implementations as described in the section 4 in the “Course Project” description.

As implied by the name of each class, it is clear to know what each class does. The only library imported in this project are “java.util.ArrayList” and “java.util.Random”. As the size of arrays in Java cannot be changed once declared, ArrayList is imported to provide an ease of array size operation as given in c++ by default. “java.util.Random” is imported to generate random number in class “GraphGenerator” and “Main”.

**Description of each class:**

1. **Class “Edge”:** Edge(int nodeA, int nodeB, int weight)

This class defines “edge”. To declare an edge, three things need to be taken in as arguments, two nodes on both ends of the edge, as well as the weight of the edge. Mostly the methods in this class are getter and setter, that does not require many complicated algorithms.

1. **Class “Graph”:** Graph(int numVertices)

This class is the most important component in this project. The major field member of this class is adjListMap, which is an array of ArrayList of edges (i.e. adjacency list); adjListMap[i] represents a list a edges of vertex i. To return an edge set of a vertex, do "adjListMap[i]". Besides its major field, it also contains the most fundamental graph operation, addEdge(), which add edges implemented in the class “Edge”, to adjListMap. Plus, the three “MAX-BANDWIDTH-PATH” algorithms are also implemented in this “Graph” class. These three algorithms will also be described and analyzed separately in one of the following sections in this report. This class has its own tester in its own main method. Graph can be generated and be printed out as desired.

1. **Class “GraphGenerator”:** GraphGenerator()

This class is implemented to generates graphs. By calling the constructor, it initializes the random number generator. It contains two main methods that are not private, generateSparseGraph() and generateDenseGraph().

Graph generateSparseGraph(int numVertices, int avgDegreeNum\_predefined)

Graph generateDenseGraph(int numVertices, int degreePercentage\_predefined)

By taking in the number of vertices and number of degrees or percentage of adjacencies desired, it can generate random sparse graphs or random dense graphs of as many vertices as desired accordingly (if time and space requirements are not constrained). The random weight generated in this class is set between 11 to 1000. By randomly pick the vertices to be added to each of vertex in the graph with a randomly picked weight value, edges are made and added to the graph. At the beginning of generating graphs, a connected graph on all vertices needs to be made first through generateConnectedGraph() so that the rest of the procedure of creating graphs can be built up on a connected graph. numVertices is the number of vertices desired. isCyclic, while true, will make the connected Graph a cycle, otherwise, vertices will be simply connected. In this project, isCyclic is set to true all the time.

private Graph generateConnectedGraph(int numVertices, boolean isCyclic)

While the basic concept is the same for both generateSparseGraph() and generateDenseGraph(), the details are somewhat different. In generateSparseGraph(), as required in the “Course Project” description, a sparse graph with an average degree of 6 should be generated. Therefore, both ends of the edge are randomly picked and added to the graph as long as the (total number of edges/2) is equal to (the number of vertices \* 6). In generateDenseGraph(), as required in the “Course Project” description, a dense graph in which each vertex is adjacent to about 20% of the other vertices should be generated. Hence, the number of edges that each vertex holds should be roughly the same. So edges are to be added to each vertex one by one. It is obvious that the more edges are added, the less possibility a vertex with edges lower than a certain number can be found. Therefore, some vertex has a bit more edges, some a bit less. The implementation demonstrates an average of 1005 degrees with 5000 vertices in total and 20% adjacencies requirements, which is quite close to 5000\*20% = 1000 degrees.

This class has been tested and made sure to be correct in its own main method.

1. **Class “MaxHeap”:** MaxHeap(int predefinedSize)

There are three arrays as class members in this class, verticesName[], data[] and pos[], as the max heap can be represented as an array. When initialized, all of them are of size that is equal to the number of vertices held in “Graph” instance. The class members are listed below:

int[] verticesName; //each cell stores the vertexName

int[] data; //data[i] stores the weight that corresponds to verticesName[i]

int[] pos; //ex. vertexName = 3 is stored in verticesName[5] -> pos[3] = 5;

int actualSize; //actual size of the heap

Since the class members are arrays, for which the size cannot be changed, int actualSize can be used to trace the actual size of the array. It also functions as an indexer in the MaxHeap operations. The major operations are listed below:

void insert(int vertexName, int value)

int remove(int nodeIdx)

void swap(int nodeA, int nodeB)

When inserting, removing data into, from the max heap, the heap structure should be maintained. In other words, the datum in the parent node should be always larger than its children. When inserting, the new vertex is inserted to the last position of the heap array. What follows the next is moving this new vertex up as much as possible. When removing, the vertexA to be removed is swapped with the vertexB at last position of the heap array. Thus, vertexB needs to be moved up or down and vertexA will be excluded from the heap array. In this class, helper functions such as swap(), moveUp(), and moveDown() are used to help facilitate insert and remove operations. When remove a vertex, pos[] array is used to record the position before the remove.

1. **Class “MaxSortingHeap”:** MaxSortingHeap(int predefinedSize)

This class is specifically used in the “MAX-BANDWIDTH-PATH” by Kruskal’s algorithm. It assembles the class “MaxHeap”. What differs are variable types of some class members to store “Edge” objects.

**Algorithms of finding path with maximum bandwidth:**

1. All these three algorithms below are implemented in Class “Graph”.
2. They all return maximum bandwidth they explore at the end of execution.
3. They all use constructPath to return the path.

ArrayList<Integer> constructPath(int start, int goal, int[] dad, int[] bw)

1. **Algorithm - findMBPath\_Basic:** int findMBPath\_Baisc(int start, int goal)

This graph simply uses the most basic version of Dijkstra’s algorithm. To store fringes, an ArrayList is used.

1. **Algorithm - findMBPath\_Heap:** int findMBPath\_Heap(int start, int goal)

This algorithm uses max heap as priority queue to explore the fringe with largest bandwidth, which shortens the execution time a lot as shown in the Testing section below in this report.

1. **Algorithm - findMBPath\_Kruskals:** int findMBPath\_Kruskals(int start, int goal)

This algorithm uses maximum spanning tree first and breath first search the next to find the path. The maximum spanning tree is constructed using Kruskal’s algorithm (makeSet, Find with compression, and Union), listing all edges in sequence and adding them to the tree in non-increasing order if the edge to be added doesn’t create a cycle in the tree.

**Testing Results:**

The testing results according to the Section 4 in the “Course Project” is listed below. This testing is done in Class “Main”.

This test not only demonstrates the execution time of each running, but also help make sure that all classes and algorithms are implemented correctly. Each pair of randomly generated S-T vertices are explored on Graph.findMBPath\_Baisc(), Graph.findMBPath\_Heap() and Graph.findMBPath\_Kruskals on the same graph, if the implementation of any algorithms or class have any minor glitch, the results cannot be the same. After being tested for more than 10000 times in the “test1()” and “test2()” in class “Main”, no inconsistent results were found. mainTest() in class “Main” also provides the same functions, as shown below.

Graphical user interface, text, chat or text message

Description automatically generated

mainTest() in class “Main”, as described in the Course Project, does the followings:

5 pairs of dense graphs and sparse graphs are randomly generated using class GraphGenerator. Each generated graph picks 5 pairs of randomly selected source-destination vertices. Three different find path algorithms are applied on these sets of graphs. The first 5 sets apply Find without compression. The second 5 sets apply Find with compression, along with maximum bandwidth found per run.

Execution times results are displayed as followed. Note that each set (3 runs on sparse graph and 3 runs on dense graph) adopts the same S-T vertices.

**Below without Find compression**

The running duration for No.0 Sparse Graph:

Basic testing: 702 ms

Heap testing: 110 ms

Kruskal testing: 147 ms

The running duration for No.0 Dense Graph:

Basic testing: 1890 ms

Heap testing: 1473 ms

Kruskal testing: 10881 ms

The running duration for No.1 Sparse Graph:

Basic testing: 204 ms

Heap testing: 57 ms

Kruskal testing: 54 ms

The running duration for No.1 Dense Graph:

Basic testing: 1426 ms

Heap testing: 1177 ms

Kruskal testing: 11200 ms

The running duration for No.2 Sparse Graph:

Basic testing: 214 ms

Heap testing: 44 ms

Kruskal testing: 53 ms

The running duration for No.2 Dense Graph:

Basic testing: 1546 ms

Heap testing: 1589 ms

Kruskal testing: 11053 ms

The running duration for No.3 Sparse Graph:

Basic testing: 222 ms

Heap testing: 17 ms

Kruskal testing: 51 ms

The running duration for No.3 Dense Graph:

Basic testing: 1434 ms

Heap testing: 942 ms

Kruskal testing: 11002 ms

The running duration for No.4 Sparse Graph:

Basic testing: 208 ms

Heap testing: 16 ms

Kruskal testing: 39 ms

The running duration for No.4 Dense Graph:

Basic testing: 1378 ms

Heap testing: 980 ms

Kruskal testing: 11199 ms

**Below with find compression:**

The running duration for No.0 Sparse Graph:

Basic testing: 584 ms, MaximumBW = 594

Heap testing: 99 ms, MaximumBW = 594

Kruskal testing: 198 ms, MaximumBW = 594

The running duration for No.0 Dense Graph:

Basic testing: 1323 ms, MaximumBW = 1000

Heap testing: 1657 ms, MaximumBW = 1000

Kruskal testing: 10537 ms, MaximumBW = 1000

The running duration for No.1 Sparse Graph:

Basic testing: 154 ms, MaximumBW = 735

Heap testing: 18 ms, MaximumBW = 735

Kruskal testing: 34 ms, MaximumBW = 735

The running duration for No.1 Dense Graph:

Basic testing: 1352 ms, MaximumBW = 999

Heap testing: 1432 ms, MaximumBW = 999

Kruskal testing: 10820 ms, MaximumBW = 999

The running duration for No.2 Sparse Graph:

Basic testing: 142 ms, MaximumBW = 486

Heap testing: 9 ms, MaximumBW = 486

Kruskal testing: 45 ms, MaximumBW = 486

The running duration for No.2 Dense Graph:

Basic testing: 1361 ms, MaximumBW = 999

Heap testing: 887 ms, MaximumBW = 999

Kruskal testing: 10105 ms, MaximumBW = 999

The running duration for No.3 Sparse Graph:

Basic testing: 163 ms, MaximumBW = 688

Heap testing: 14 ms, MaximumBW = 688

Kruskal testing: 43 ms, MaximumBW = 688

The running duration for No.3 Dense Graph:

Basic testing: 1320 ms, MaximumBW = 999

Heap testing: 889 ms, MaximumBW = 999

Kruskal testing: 10428 ms, MaximumBW = 999

The running duration for No.4 Sparse Graph:

Basic testing: 195 ms, MaximumBW = 755

Heap testing: 15 ms, MaximumBW = 755

Kruskal testing: 34 ms, MaximumBW = 755

The running duration for No.4 Dense Graph:

Basic testing: 1356 ms, MaximumBW = 999

Heap testing: 886 ms, MaximumBW = 999

Kruskal testing: 10347 ms, MaximumBW = 999

**Analysis:**

In total, to generate a sparse graph, 5000 vertices and 15000 edges are added. To generate a dense graph, 5000 vertices and 2500000 = 5000 \* 20% / 2 \* 5000 edges are added.

What is already known is:

1. Time complexity of “findMBPath\_Basic” is O(n^2)
2. Time complexity of “findMBPath\_Heap” is O(mlogn)
3. Time complexity of “findMBPath\_Kruskals” is O(mlogn)

From the results, it is easy to observe that the findMBPath\_Heap always takes the least time, regardless the dense graph or the sparse graph, because the max heap shrinks the time of finding fringe with maximum bandwidth is logn.

For sparse graph, the performance of the algorithm is as follows:

findMBPath\_Heap > findMBPath\_Kruskals > findMBPath\_Basic

For dense graph, the performance of the algorithm is as follows:

findMBPath\_Heap > findMBPath\_Basic > findMBPath\_ Kruskals

Obviously, in the dense graph, “findMBPath\_Kruskals” takes less time than the “findMBPath\_Basic” does, while in the sparse graph, “findMBPath\_Kruskals” takes longer time than the “findMBPath\_Basic” does, which is not consistent with the worst-case time complexity. One possible reason is that the “findMBPath\_Kruskals” will require much longer time to sort the edges as the dense graph has way more edges, which makes each run approach the worst case complexity. As “findMBPath\_Kruskals” takes too long compared to the other two, it is uncertain that if the find with compression version get the algorithm any improvements.

**possible further improvement:**

The HashSet and HashMap can be applied in various location, so that the time to find data can be reduced to O(1) from O(logn) or even O(n).