Advanced Data Structures

Project-1

Name: Tanmay Garg UFID: 3583-5781 E-mail: tgarg@ufl.edu

Java Library: JRE System Library 1.7 (JavaSE-1.7)

Compiling and executing the program:

- javac mst.java
- java mst (arguments)

1. For random graph: java mst –r [vertices] [density]

   E.g.: java mst –r 1000 50

2. For input from file: java mst –s [filename] //for simple prim's algorithm

   java mst –f [filename] //for Fibonacci heap prim's algorithm

   E.g.: java mst –s sample.txt

Structure of the program:

The program consists of 8 java classes:

1. Vertex: The vertex class represents one vertex of the graph and has attributes such as id and the list of the neighbor objects of the vertex. This is used to implement the adjacency list in which every vertex has the details of its neighbors.

2. Neighbor The neighbor class represents the neighbors of the vertex. It has a vertex number and a weight and a next pointer pointing to the next neighbor of the vertices. Every vertex has a neighbor object adjList and every neighbor points to the next neighbor of the vertex.

3. mst The mst class is the main class of the project with the main function and it calls all the functions and main objects of the GraphRandom and GraphInput class. Both of these classes extends this class.

   Functions:

   Prim(): This function initializes the matrix used for computing the simple prim's algoritm, and keeps the track of the minimum spanning tree in an array. It calls the computePrim() function for main computation.

ComputePrim(): It computes the prim's algorithm recursively by selecting the min cost using the simple approach.

shortestPaths(): It creates an object of the FHeap class, i.e. a Fibonacci heap and computes the cost of the minimum spanning tree.

Print(): It prints the graph generated either randomly or by the input file. It is used for debugging.

4. GraphRandom — This class generates a random graph according to the given number of vertices and the density and checks the graph for connectivity using dfs.

Functions:

Random(int): Generates a random number from 0 to the number passed as argument.

checkConnected(int,int): traverses a graph using dfs and checks if the graph is connected.

alotNeighbors(int,int): allots the neighbors to the vertices, i.e. creates the edges in the graph.

5. GraphInput — This class reads the input from the file and generates the graph according to the given specifications.

6. Edge — This class contains two variables for vertices and one cost and it is used to calculate the cost of mst using basic prim's algorithm.

7. FHeap — This class contains the structure of the Fibonacci heap and all the operations related to that structure required in calculating the minimum spanning tree using the Fibonacci heap.

Functions:

Enqueue(int,double): Inserts the vertex with cost into the Fibonacci heap

DequeueMin(): Removes the min element from the heap

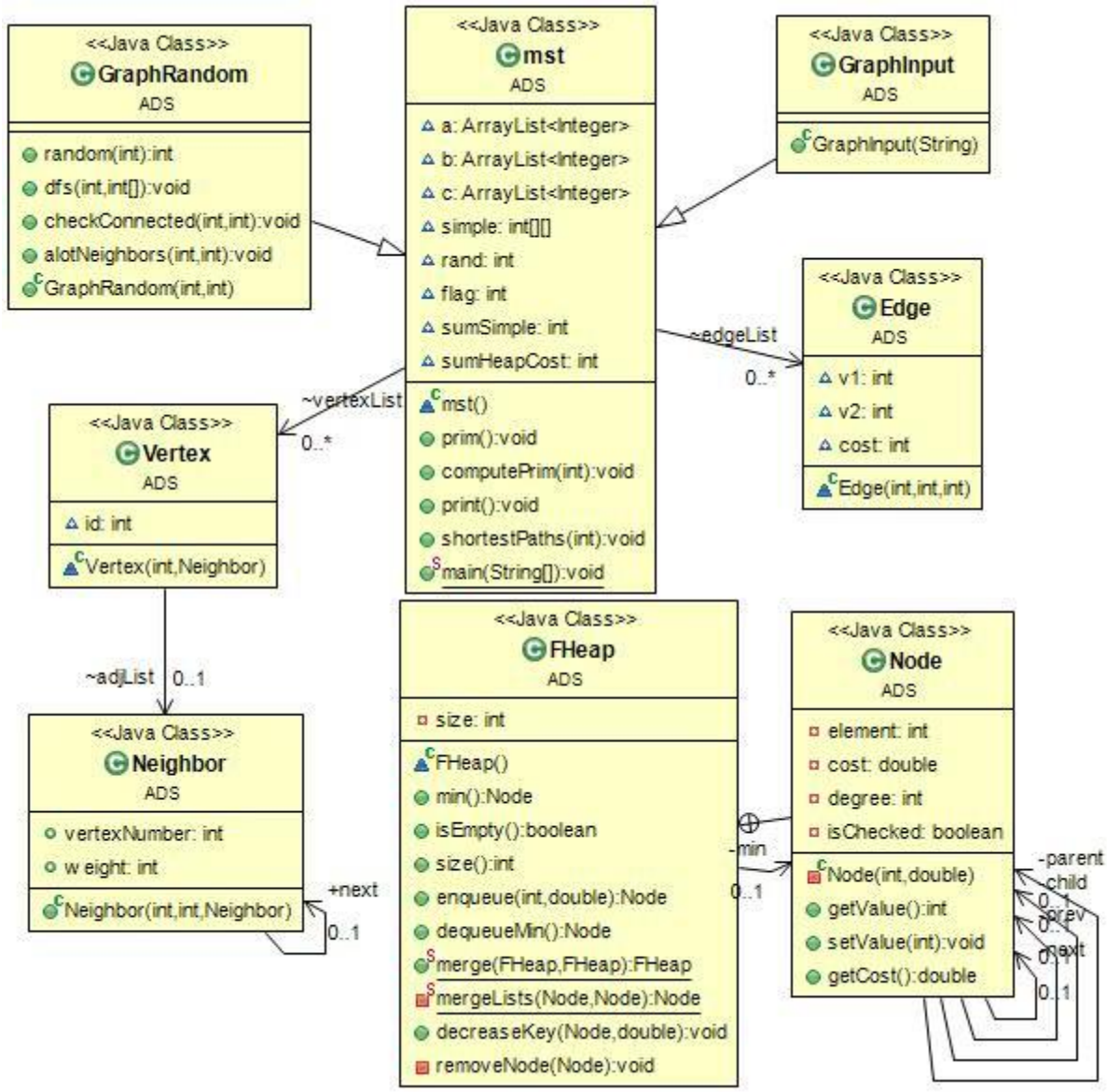
Merge(FHeap,FHeap): Merges the two Fibonacci heaps

mergeList(Node, Node): Merge the two lists together into one circularly-linked lsit

decreaseKey(Node, Double): Decreases the key of the specified element to the new cost and check if new cost is less.

8. Node — Node class represent one node of the fibonacci heap and consists of element, cost, degree and isChecked to check if marked.

Below is a detailed UML diagram showing the structure of the program with all the class definitions and the function prototypes.



Expectation of Result Comparison:

For the larger number of vertexes i.e. bigger graphs, the Fibonacci heap should work better for the less number of edges i.e. less density, and the simple approach should work better for higher densities. For less vertexes, i.e. smaller graphs, simple approach should be more efficient.

This should be because fibonacci works with a complexity of O(vlogv+e), where v is the number of vertices and e is the number of edges. As the density increases, the number of edges increase, and become the dominating cost factor.

Execution Time Table:

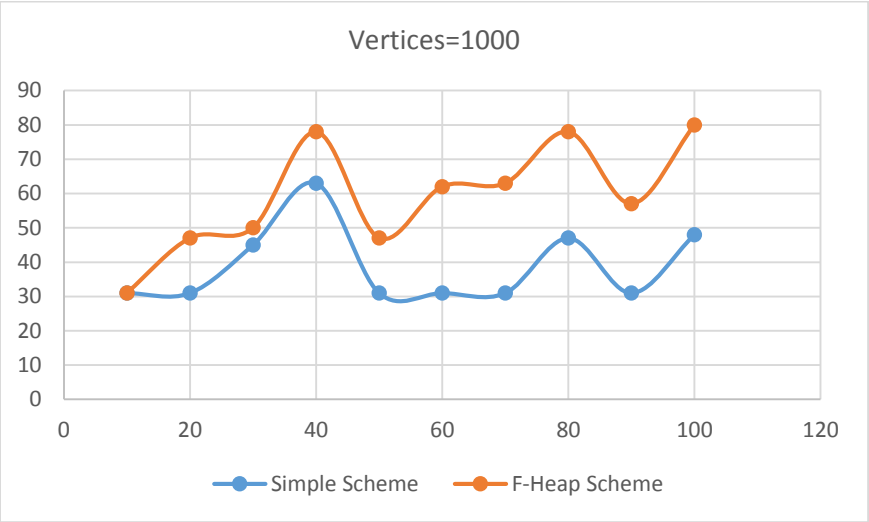
N (Number of Vertices)	Density	Simple Scheme	F-Heap Scheme
1000	10	31	31
	20	31	47
	30	45	50
	40	63	78
	50	31	47
	60	31	62
	70	31	63
	80	47	78
	90	31	57
	100	48	80

N (Number of Vertices)	Density	Simple Scheme	F-Heap Scheme
3000	10	94	93
	20	94	125
	30	203	188
	40	170	219
	50	140	235
	60	266	344
	70	203	329
	80	224	260
	90	190	342
	100	200	450

N (Number of Vertices)	Density	Simple Scheme	F-Heap Scheme
5000	10	281	156
	20	188	265
	30	282	406
	40	284	568
	50	391	2312
	60	400	3012
	70	547	3703
	80	632	4552
	90	734	4234
	100	802	6574

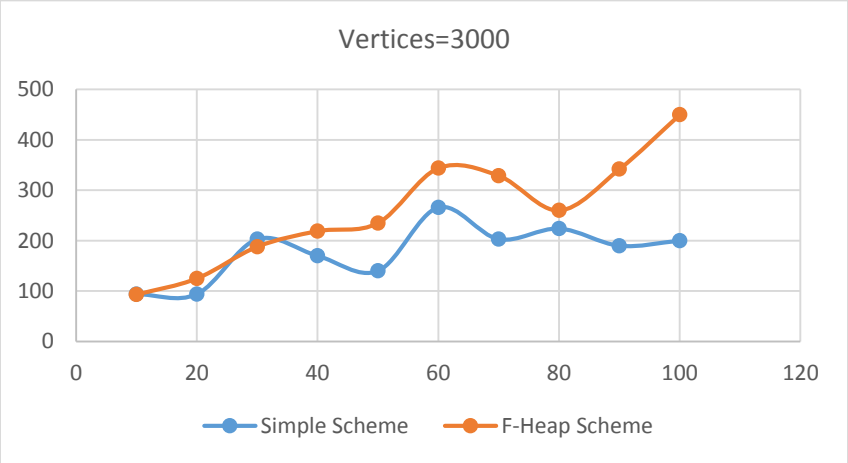
Graphs based on the values:

For n=1000



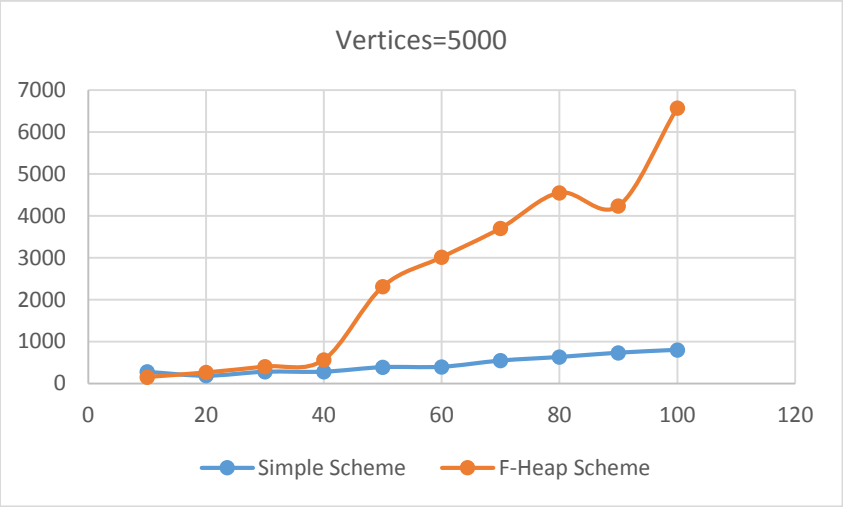
For number of vertices=1000, the simple scheme is always more efficient than the F-Heap scheme. And as per my analysis, the time for F-Heap goes on increasing with the increasing number of edged i.e. increasing density, whereas it was same for the initial density.

For n=3000



For the number of vertices=3000, the simple scheme and F-heap scheme is same for the initial 10 density, but as the density goes on increasing the F-heap approach starts taking more time.

For n=5000



For number of vertices=5000, the F-heap scheme is more efficient than the simple scheme which is correct according to my analysis that for larger number of vertices and less density the F-heap approach will be more efficient than the simple approach. But as the density goes on increasing, the F-heap keeps taking more time than the simple because of the increased number of edges.

Thus my analysis was somewhat true that for larger number of vertices and less density, the F-heap will outperform the simple approach but as we increase the density, the simple will be the better one. I didn't analyze the bumps which are shown in the graphs and I think those might be related to the processor caching as the bumps came on different values for different type of processors.