REPORT

NAME : MONICA MAVOORI
UFID : 46309228
EMAIL ID : mmavoori@ufl.edu

EXECUTION STEPS:

Java compiler is used to run the files. To execute the files go the path where the files are present and follow the below order.

- For User Input Mode:
  - Compile the .java file as ***javac MST.java***
  - Now, run Simple Scheme type the command as
    java MST –s inputfilename
  - To run F-heap scheme type the command as
    java MST –f inputfilename
  - Due to virtual machines max heap size we may get an exception OUTOFMEMEORY EXCEPTION: JAVA HEAP SPACE, then run the command as
    java -Xms256m -Xmx4096m MST –s(or -f) inputfilename

- For Random Mode:
  - Compile the .java file as ***javac MST.java***
  - Now, run the file as
    java MST –r numberofvertices density
  - Due to virtual machines max heap size we may get an exception OUTOFMEMEORY EXCEPTION: JAVA HEAP SPACE, then run the command as
    java -Xms256m -Xmx4096m MST –r numberofvertices density

STRUCTURE OF THE PROGRAM AND LIST OF FUNCTIONS USED:

Classes and its Methods:

- Class MST

  This class contains the main method of the program. It takes the arguments file name for simple scheme and number of vertices and density for f-heap Scheme and calls the particular parameterized constructor of Graph class to create the graph according to the inputs provided. If the arguments given are not sufficient for any of the scheme it prints a message for insufficient no of arguments.

- Class Edge

  This is a support class for the Graph class to build edges. Each edge has a number, Its neighbor edge and cost associated to it.

- Class Vertex

  This class is used for the vertices of Graph. Each vertex is associated to vertex number, its edges list and a boolean to mark its visit.

- Class Graph
  - Parameterized constructor Graph(String filename)

    This constructor is called when the input given is a file consisting of data for graph creation. It takes the information from input file and builds the graph accordingly. It ends when reaches end of the file.
  - Parameterized constructor Graph(int no_of_vertices,int density)

    This constructor accepts two integer variables and the input given is no of vertices with which we build the graph and density. The edges are randomly created until given density is reached and the graph thus constructed is always checked whether it is connected by calling dfs() method.
  - Method boolean dfs()

    This method checks whether the graph constructed is connected or not by visiting each vertex and writing it on to stack if it has children. Returns a boolean true if the vertex on stack is visited or else returns false.
  - Method int getUnvisitedVertex(int n)

    This method returns an integer i.e. vertex number which is the not visited child of input vertex n.
- Class Prims
  - Parameterized constructor Prims(Graph p)

    This constructor is for the simple scheme where the time complexity is O(n2). It calls the check method to calculate the least cost edges.
  - Method Check(Vertex[] adjLists)

    This method takes the graph's adjacency lists as parameter and checks for least cost edge passing from a vertex. It makes sure not to add a vertex which is already visited.

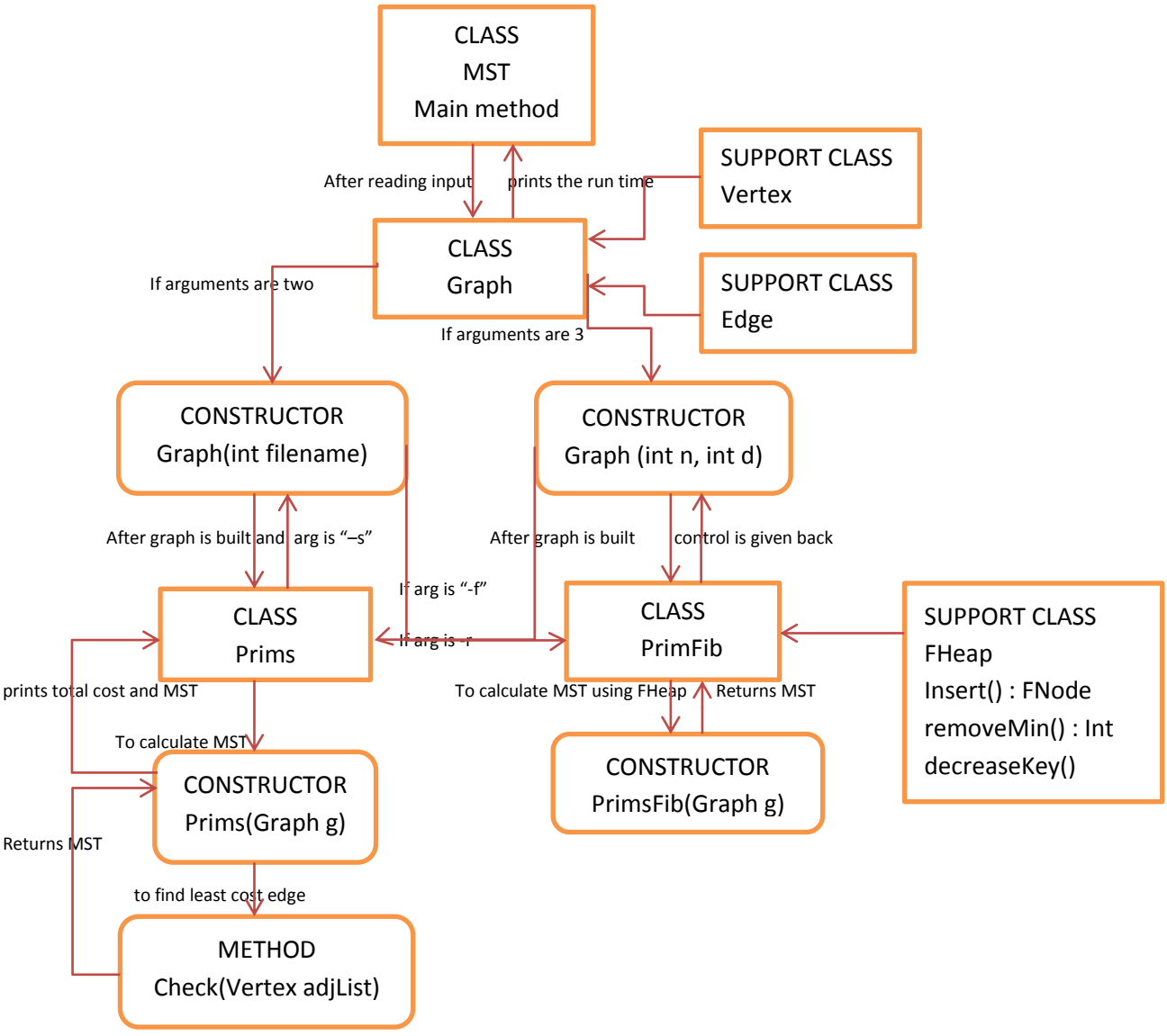
    By checking each and every neighbor vertex's cost it choses lease cost edge and that to the result graph until MST is formed.
- Class PrimFib
  - Parameterized constructor PrimFib(Graph p)

    This constructor is for f-heap scheme which takes O(E + VlogV) time to construct MST from an input graph. It calls calculate method to find least cost edges. It uses the methods of f-heap class like removeMin(), insert() and decreaseKey() to calculate the least cost edge until an MST is formed.

RESULT OF COMPARISON:

Adjacency list is the most efficient way to represent a graph. It is an array of linked lists. Prims algorithm is implemented using simple scheme and f-heap scheme in the source code. In simple scheme the one for loop runs for V number of times where V represents number of vertices given to construct a graph. For each vertex there must be another for loop to run for all its neighbors. Therefore in worst case the second for loop also runs for V times. Total time complexity of the program in simple scheme is O(n2). In second scheme, i.e. f-heap scheme it used a data structure Fibonacci heap to construct MST. In this scheme a heap operation takes log V steps. Total time complexity to find MST using Fibonacci data structure is O(E + VlogV) where E is the number of edges. For sparse graphs Fibonacci scheme is the most efficient one to use to find their MST's, as it saves much time.

FLOW OF THE SOURCE CODE:



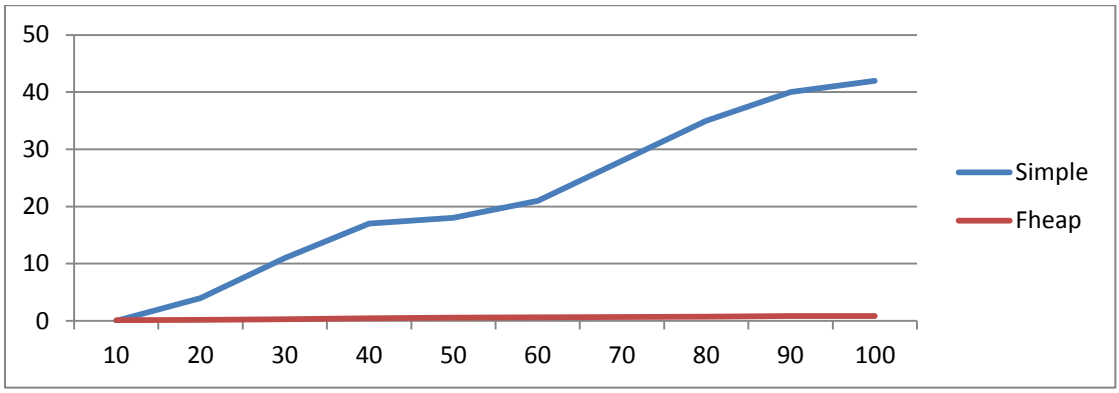
GRAPHICAL REPRESENTAION OF THE ALGORITHM PERFORMANCE IN SIMPLE AND F-HEAP SCHEME:

Below are the graphs plotted comparing the runs time of Simple scheme and f-heap scheme:

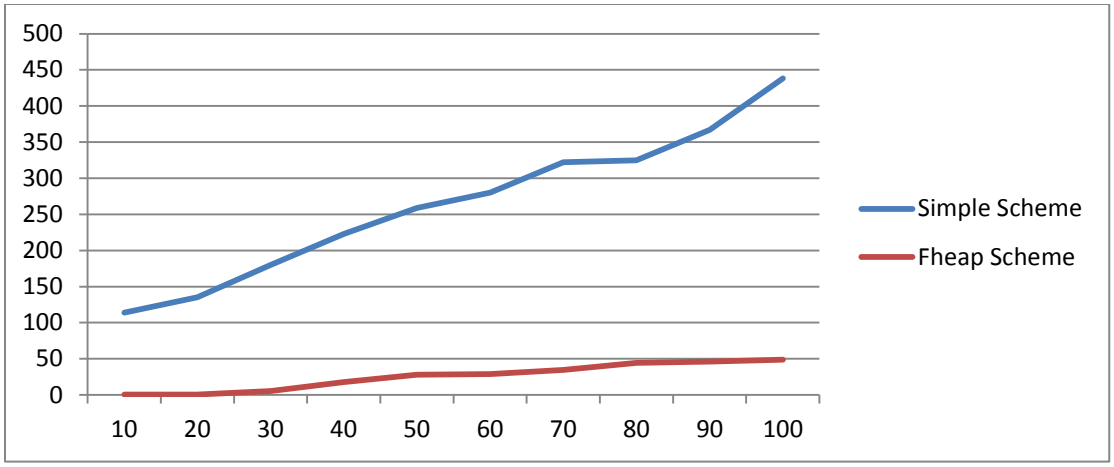
X-axis : Density of the graph (%)

Y-axis : Time taken to run the program (sec)

For 1000 Vertices:



For 3000 vertices:



For 5000 Vertices:

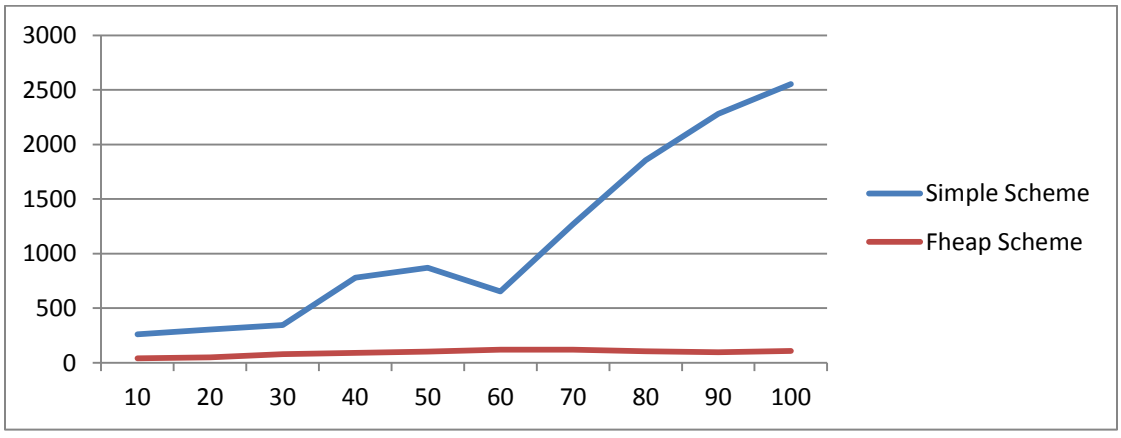


TABLE INDICATING THE VALUES OF TIME OF EXECUTION FOR BOTH THE SCHEMES (SIMPLE AND F-HEAP) IN SECONDS WITH NO OF VERTICES AS 1000,3000 AND 5000 AND INCREASING DENSITY.

DENSITY (%)	TIME FOR EXECUTION(IN SEC)					
	No of vertices(1000)		No of vertices(3000)		No of vertices(5000)	
	Simple Scheme	FHeap Scheme	Simple Scheme	FHeap Scheme	Simple Scheme	FHeap Scheme
10	0	0.12	114	0.62	260	40.7
20	4	0.18	135	0.69	304	49.3
30	9	0.27	180	5.60	344	78.9
40	14	0.44	223	17.8	778	91.1
50	19	0.57	259	28.1	870	101.3
60	23	0.62	280	29.1	654	119.5
70	28	0.69	322	34.5	1269	120.7
80	32	0.75	325	44.2	1854	104.3
90	37	0.84	367	46.3	2281	94.9
100	42	0.88	438	48.9	2554	107.4

Above table and the graphs clearly identifies the efficiency of f-heap scheme. F-heap scheme saves much time when compared to simple scheme in calculating MST when a connected graph is provided. When number of vertices increases time of execution also increases, so for sparse graph simple scheme takes much higher time when compared to f-heap scheme. In real time applications where time is the important constraint to consider, using f-heap scheme will be efficient.