Report

Name: Kiran Vasudeo Rohankar

UFID: **91996253**

Email Id: kvr130030@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, for simple scheme run the file as ***java Mst –s inputfilename***
  - Now, for f-heap scheme run the file as ***java Mst -f inputfilename***

- For Random mode
  - Compile the .java file as ***javac Mst.java***
  - Now, run the file as
    java Mst -r numberofvertices density

Description of classes and Functions:

1) Mst()

- **Mst(int, int, Vertex[]):**This is the parameterized constructor which initializes no of vertices and edges in the class. It also initializes array of vertexes where each vertex is a instance of class Vertex
- **Mst():**This is the default constructor which sets value of vertex and edges to zero and makes array of vertex null.
- **createGraph(String):**This function reads the input from a given text file which is provided by its parameter. Line by line it adds edge to adjacency list. At the end of the file it will object of the created graph.
- **print():** It just prints the graph as an edge and cost pair.
- **main(String[]):**This is the main function which decides the flow of the program.

2) RandomGraph()

- **Random_Graph(int, int):** This is the parameterized constructor which initializes the vertices and edges depends on density provided by the main function.
- **createGraph():** This function will created the random edges between any two random vertices. It assigns random weight between 1 to 1000 to the edges.It takes care that the

graph is connected by calling depthFirstSearch(randomGraph).If it returns false then it keep on adding the edges till the graph is connected.

- **depthFisrtSearch(randomGraph):**This function takes the randomGraph instance as a input and then returns true if the all the vertices in the graph are connected else it returns false.
- **print():**It just prints the graph as an edge and cost pair.

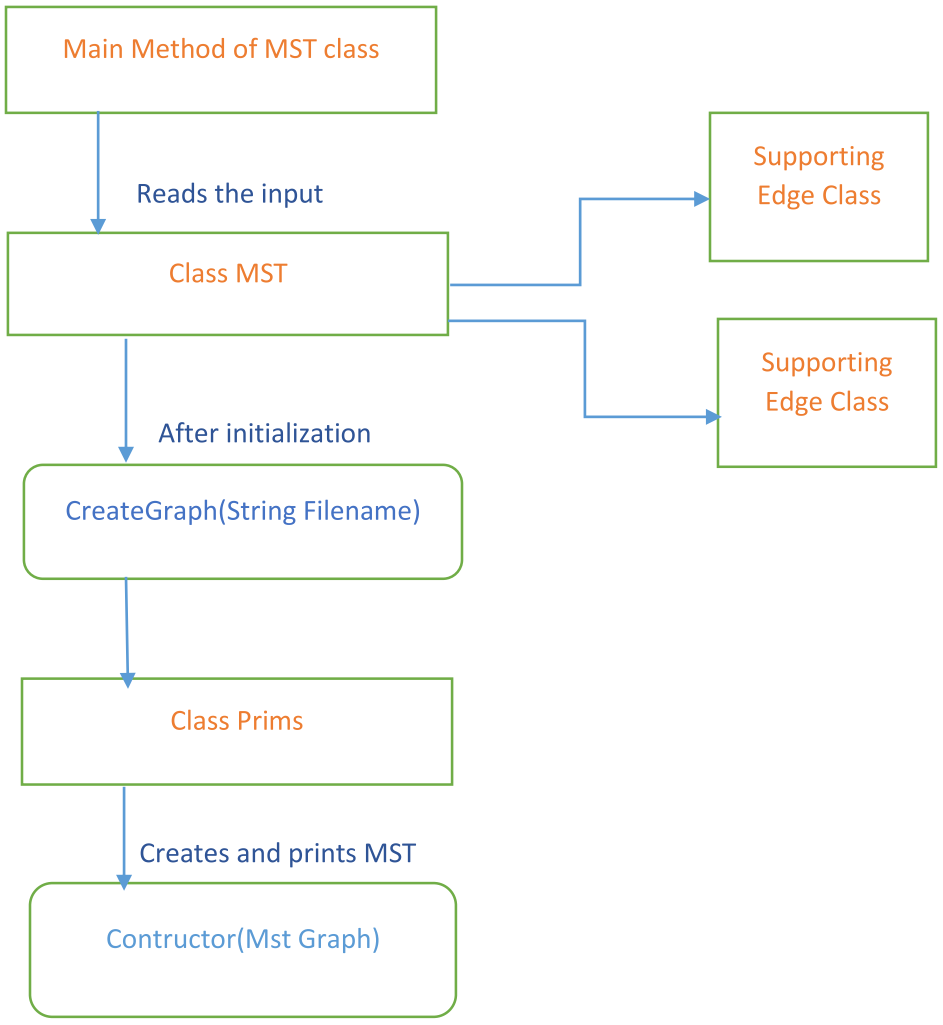
3) **Edge()**
    - **Edge(int, int, Edge):**This is a parameterized constructor for edge class. It initializes the associated vertex number and cost with the edge. It also has the reference pointing to next edge creating a linked list. This will be beneficial for creating adjacency list so that we can know neighbors of the vertices.
    - Edge():This is a default constructor for an edge class.
4) **Vertex()**
    - **Vertex(int, Edge, boolean):**This is a parameterized constructor for an vertex class. It initializes the identity for current vertex. It will have edge for which the vertex is associated and third Boolean variable as visited.
    - Vertex():This is the default constructor for a vertex class.
5) **Prims()**
    - **Prims(Random_Graph):**This is the overloaded constructor which takes random graph instance and calculates the minimum spanning tree.

6) **FibonacciHeap()**
    - **FibonacciHeap():**Default constructor of fibonacciHeap class.
    - **isEmpty():**returns true to heap is empty.
    - **clear():**sets heap to its default empty state.
    - **insert(FibonacciHeapNode, double):**This function takes Fibonacci node for and key as a cost of that node. It insert that node in the Fibonacci heap.
    - **min():**It returns minimum element in the heap.
    - **size():**It returns size of an heap.
    - **removeMin():**It returns minimum element in the heap. After removing minimum elements it re-arranges heap by calling consolidate function. Consolidate function then merges same degree trees.
    - **decreaseKey(FibonacciHeapNode, double):** This function has the parameter as reference of node and key of the node. If the current key is bigger than the key in the parameter decrease key function sets new key to the current key.
    - **prims_Fhip(Random_Graph):** This function calculates minimum spanning tree of the given graph using Fibonacci heap.

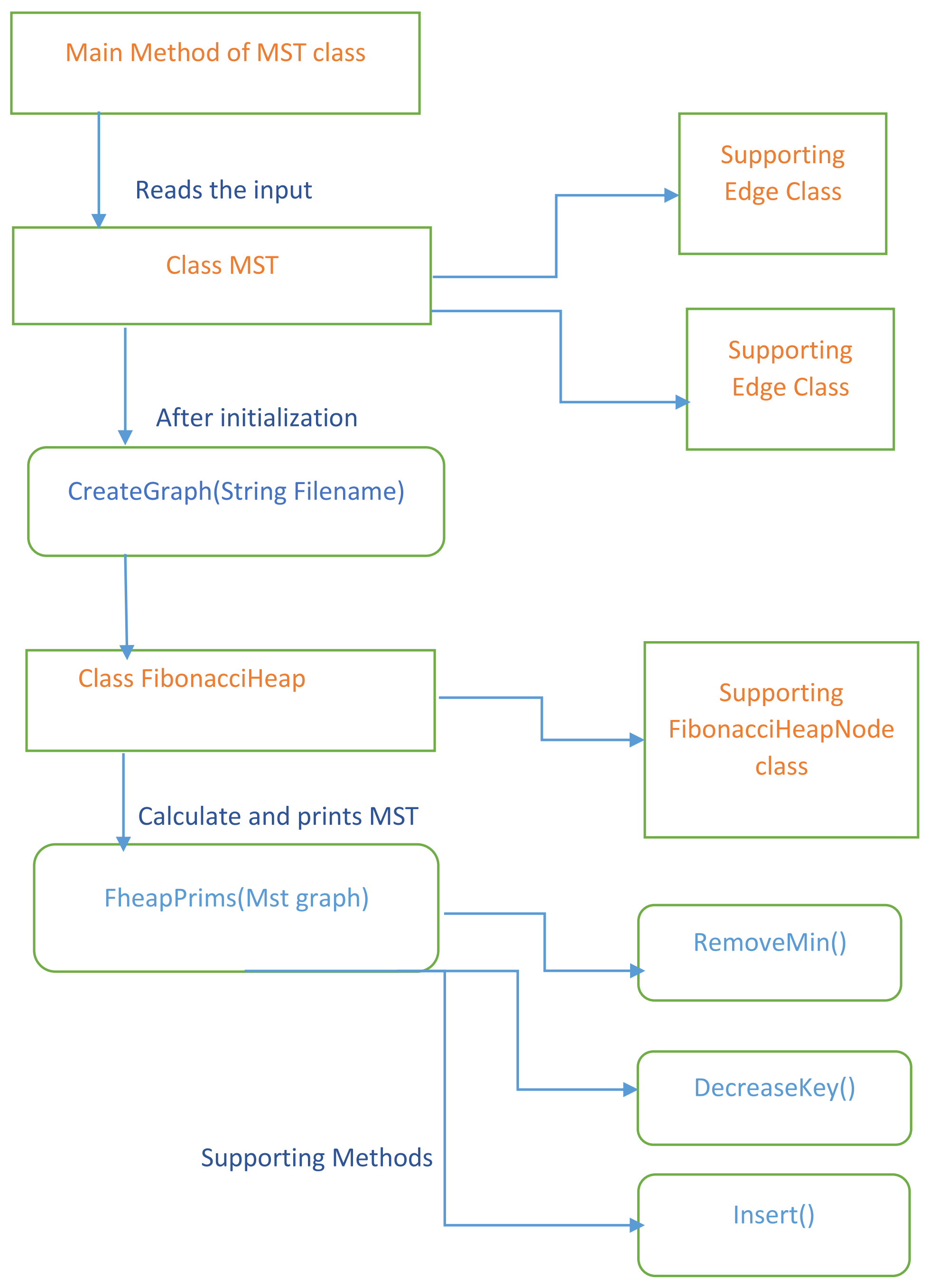
7) **FibonacciHeapNode()**
    - **FibonacciHeapNode(int, double):**It is a parameterized constructor which initializes the node of FibonacciHeapNode.
    - **getKey():**returns key associated with the node.

Structure of a program

Simple scheme with file input

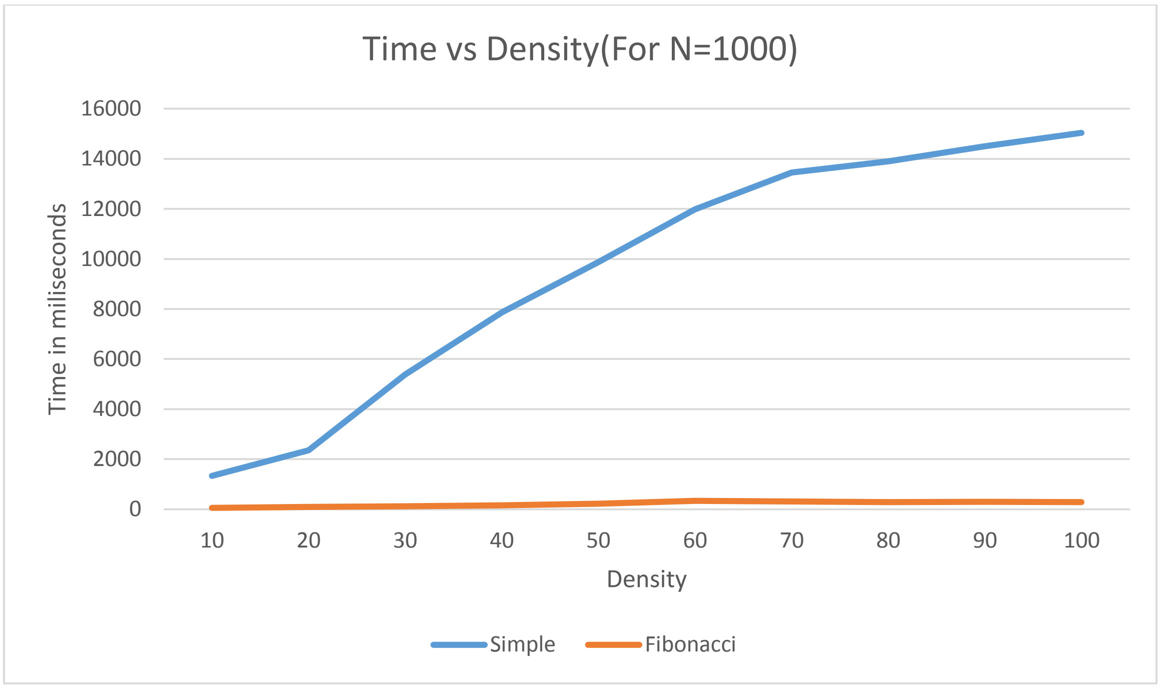


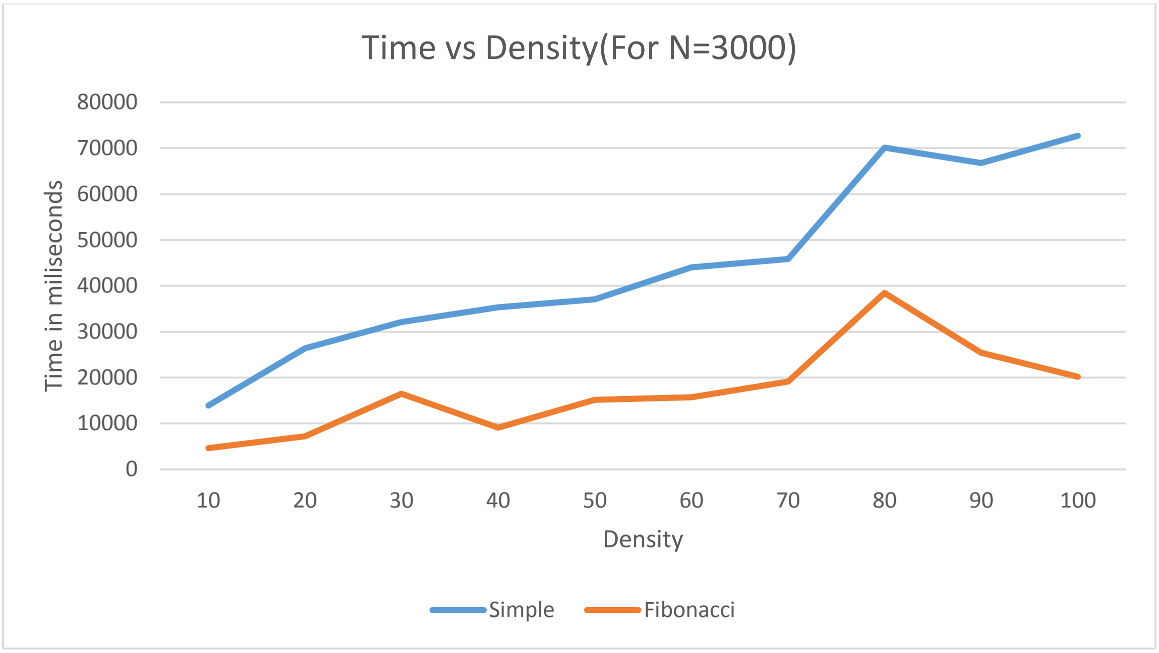
Fibonacci scheme with file input



Note : Similar flow for random graph using f-heap and adjacency lists.

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





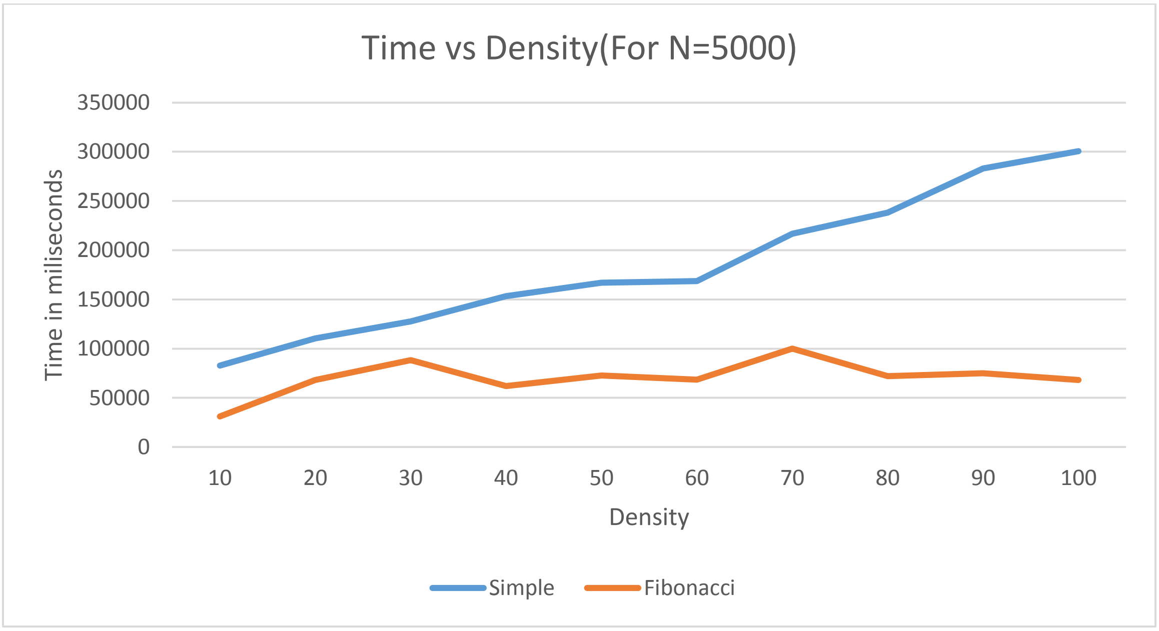


TABLE INDICATING THE VALUES OF TIME OF EXECUTION FOR BOTH THE SCHEMES (SIMPLE AND F-HEAP) IN SECONDS WITH NO OF VERTICES AS 1000,3000 AND 5000 AND INCREASING DENSITY.

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	1336	51	13880	4634	82717	31039
20	2354	94	26360	7173	110469	68280

30	5382	115	32094	16444	127703	88171
40	7863	156	35334	9079	153240	61826
50	9876	216	37049	15146	167039	72861
60	11982	280	44015	15723	168552	63485
70	13452	329	45858	19141	216782	100148
80	13900	314	70139	38421	238305	72101
90	14509	280	66812	25403	282898	75129
100	15038	299	72708	20167	300598	68056

Conclusions:

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. In real time applications where time is the important constraint to consider using f-heap scheme will be efficient.