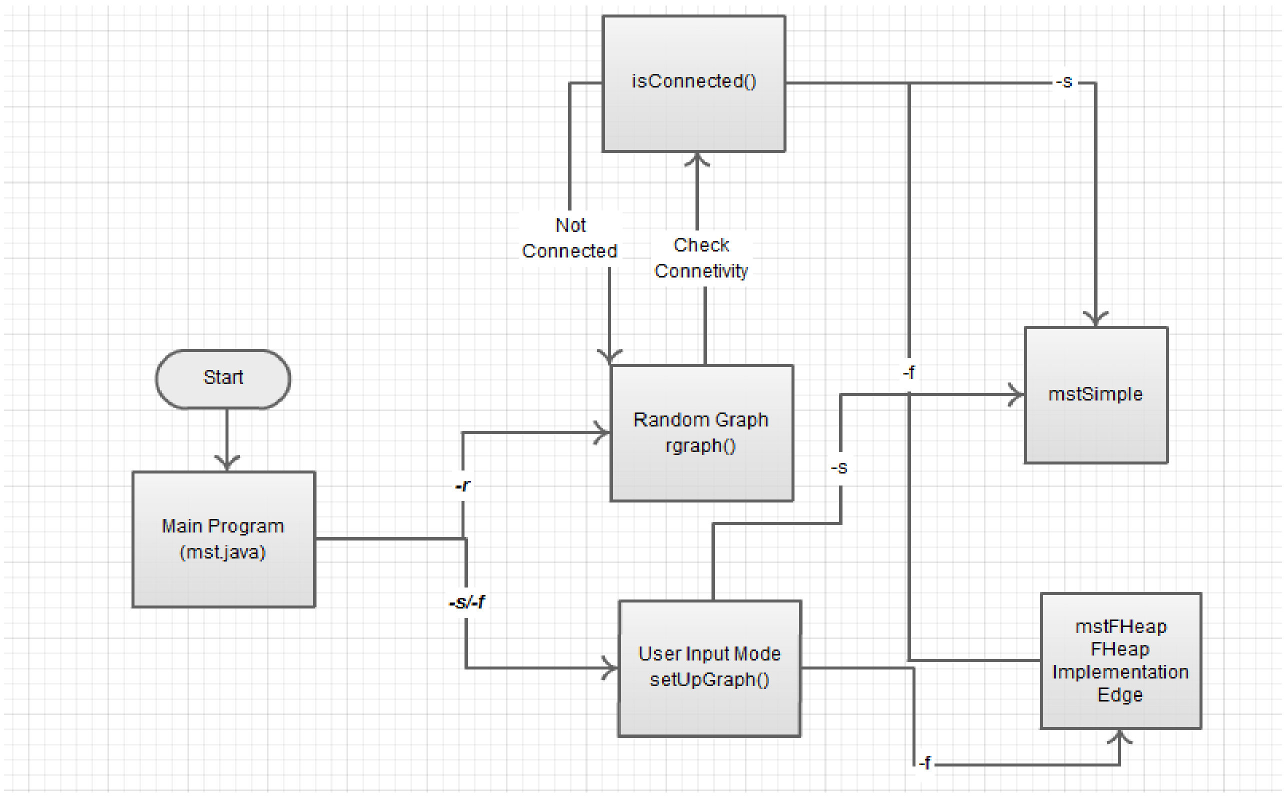
I. How to compile and run?

Compiler Used: JVM
Compile: javac mst.java
Run: java mst.java –r n d
Or java mst.java –s filename
Or java mst.java –f filename

Please note: To run be inside Jain_Akhil folder i.e LastName_FirstName folder.

II. Program flow and Function Prototype*:*



The project package consists of 4 files:

- mst.java: Contains main function and all other important function.
- Edge.java: Contains Edge class which is basically used for getter and setter of edges between vertices.
- FHeap.java: Actual Fibonacci Heap implementation.
- Node.java: Create Heap nodes.

<u>mst.java</u>

public class mst
Actual starting point of the program that takes user input modes in the main method.
-r mode for Random Mode in which we form a random graph and check if connected then use simple scheme and f-heap scheme and compare their run time.
-s mode uses file input and form a graph using simple array and then print MST.
-f mode uses file input and form a graph using f-heap and then print MST.

private void rGraph(int density)
Parameter: Density of the graph
Generates a Random graph. Function generates random edges and cost and check if that edge is present in the edgeList. If no then add else discard. The process continues till all desired vertices are linked i.e. graph is connected.

private void mstSimple (String mode)
Parameter: Run mode
Generates MST using simple scheme. Process consists of decrease key and extract minimum.

private void mstFHeap (String mode)
Parameter: Run mode
Generates MST using f-heap scheme. Process consists of decrease key and extract minimum using Fibonacci heap.

private boolean isConnected()
Return a boolean stating whether graph is connected or not.Graph connectivity is checked using Depth First Search (DFS) algorithm.

Edge.java

public Class Edge
Defines an edge between 2 vertex and a cost is associated with it. Usual getter and setter are included.

FHeap.java

public final class Fheap<T>
A class representing a f-heap scheme.

public Node<T> enqueue(T value, double priority)
Inserts the specified element into the Fibonacci heap with the specified priority.

public static <T> Fheap<T> merge(Fheap<T> one, Fheap<T> two)
Parameter one: first Fibonacci heap to merge.
Parameter two: second Fibonacci heap to merge.
Return: A new Fibonacci Heap containing all of the elements of both heaps.

public Node<T> dequeueMin()

Return: The smallest element of the Fibonacci heap.

public void decreaseKey(Node<T> Node, double newPriority)

Parameter: Node The element whose priority should be decreased.

Parameter newPriority The new priority to associate with this Node.

Decreases the key of the specified element to the new priority.

private void cutNode(Node<T> Node)

Parameter: Node The node to cut from its parent.

Cuts a node from its parent. If the parent was already marked, recursively cuts that node from its parent as well.

Node.Java

Node(T elem, double priority)

Constructs a new node that holds the given element with the indicated priority. Node class consist of various attributes like #children, next & previous in the list, parent & child in the list, element, priority.

III. Comparison of Schemes:

Ideal Case:

Time complexity for both the schemes is as follows:

a) Simple Scheme: It takes O(V^2) time as for each vertex, distance array is maintained and accessed to find the minimum one (V = number of vertices). Also, to update distances it takes total O(V^2) time. Therefore, overall complexity is O(V^2).

b) F-Heap Scheme: It takes O(VlogV + E). Min edge is found n-1 times and it takes logv time to find one such vertex. Hence VlogV. Since decrease key is done for each edge it takes O(E) time.

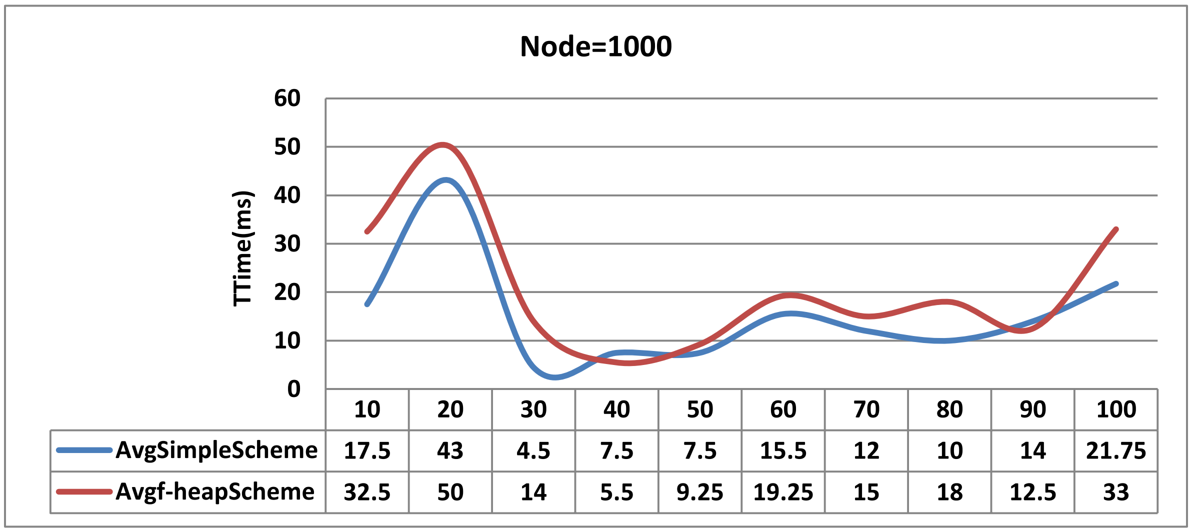
Now important thing to note is in a dense graph E=O(V^2).Both the schemes should perform similarly asymptotically for a dense graph. But for a sparse graph F-Heap Scheme should perform better in ideal situations.(*Please note the above case also signify my expectation*).

Practical case:

Implementation of above two schemes was carried out in random mode with number of vertices being 1000, 2000, 3000 and 5000. Each case is handled with varying density 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%, 90% and 100%. The analyses are as follows:

AvgSimpleScheme =	Average taken over 5 times for each varying Density for Simple Scheme
Avgf-heapScheme =	Average taken over 5 times for each varying Density for f-Heap Scheme

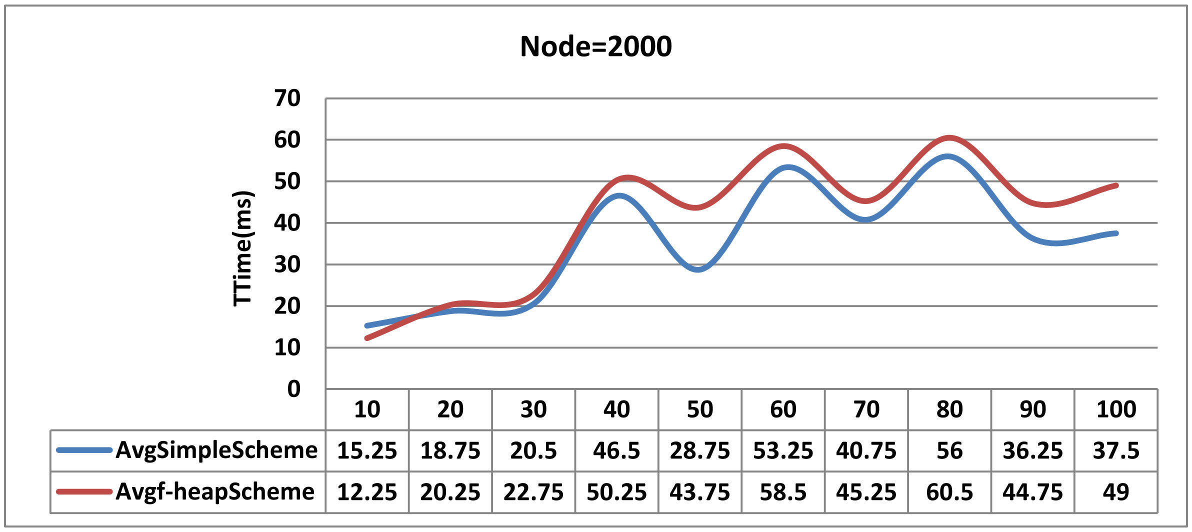
Case1: n=1000 with varying density



	10	20	30	40	50	60	70	80	90	100
AvgSimpleScheme	17.5	43	4.5	7.5	7.5	15.5	12	10	14	21.75
Avgf-heapScheme	32.5	50	14	5.5	9.25	19.25	15	18	12.5	33

***Key observation*:**
Simple scheme works better than f-Heap scheme for almost all densities. It may be because of the overhead incurred by f-Heap Scheme (due to many pointers in its implementation) where as simple scheme has relatively simpler implementation without overhead.

Case2: n=2000 with varying density

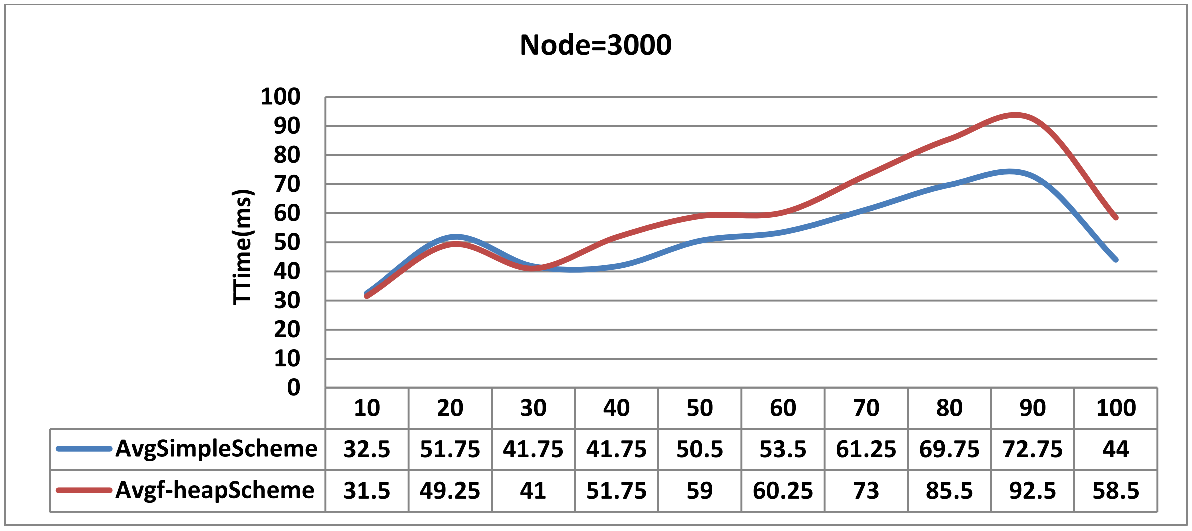


	10	20	30	40	50	60	70	80	90	100
AvgSimpleScheme	15.25	18.75	20.5	46.5	28.75	53.25	40.75	56	36.25	37.5
Avgf-heapScheme	12.25	20.25	22.75	50.25	43.75	58.5	45.25	60.5	44.75	49

***Key observation*:**

It can be seen that f-Heap works better than simple scheme until approximately 10 % density. As we approach density=100%, simple scheme works better than f-Heap scheme which is evident as both are of O(n^2) for a dense graph but simple scheme works better because of less overhead.

Case3: n=3000 with varying density

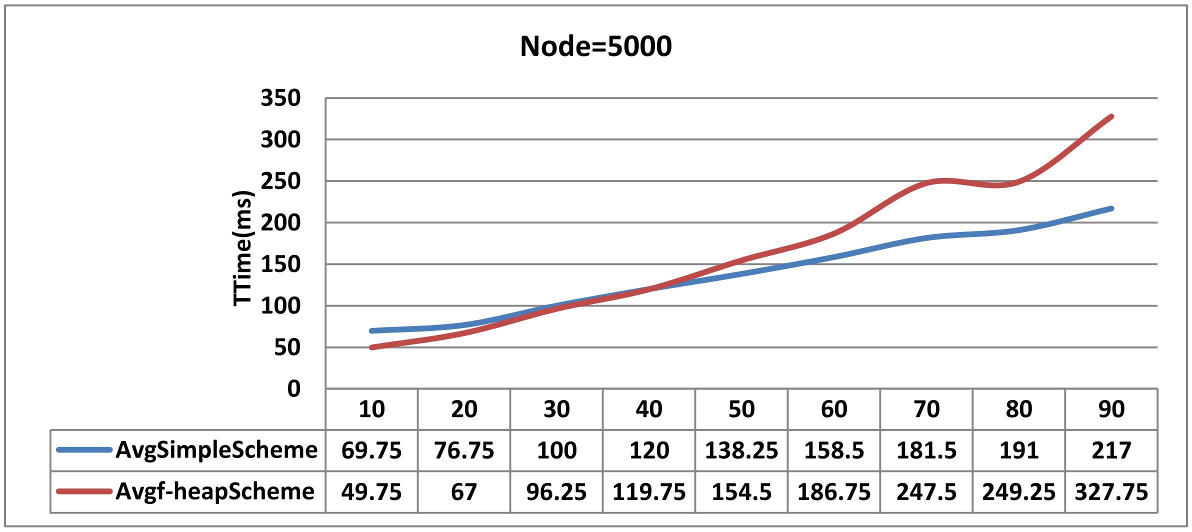


	10	20	30	40	50	60	70	80	90	100
AvgSimpleScheme	32.5	51.75	41.75	41.75	50.5	53.5	61.25	69.75	72.75	44
Avgf-heapScheme	31.5	49.25	41	51.75	59	60.25	73	85.5	92.5	58.5

Key observation:

Now the number of Vertices have been increased to 3000. It can be seen that f-Heap works better than simple scheme until approximately 30 % density. As we approach density=100%, simple scheme works better than f-Heap scheme which is evident as both are of O(n^2) for a dense graph but simple scheme works better because of less overhead.

Case4: n=5000 with varying density



	10	20	30	40	50	60	70	80	90
AvgSimpleScheme	69.75	76.75	100	120	138.25	158.5	181.5	191	217
Avgf-heapScheme	49.75	67	96.25	119.75	154.5	186.75	247.5	249.25	327.75

Key observation:

A similar trend of f-Heap performing better than simple scheme is observed for sparse graphs. It works better until 40% density and then simple schemes performs better for n = 5000.

IV. **Conclusion**:

As discussed in class remove minimum and arbitrary remove work in O(log n) amortized time. This means that starting from an empty data structure, any sequence of a operations from the first group and b operations from the second group would take O(a + b log n) time. For sparse and less number of node, the constant associated with it is might be higher. For smaller nodes and lesser density(i.e. small n value) simple heap outperforms f-heap scheme due to the additional overhead involved in f-heap .But for higher node like 8000, 10000 and so on, the time difference between simple scheme and f-heap significantly increased (for n=10000 density=10% simpleSchemeTime=851 ms and f-heapScheme=608 ms). This clearly signifies that for higher node and sparse graph f-heap outperforms simple scheme. But as far as dense graph are concerned it is evident that both perform in a similar fashion.

PLEASE Note: Performance reading and graph plot is also available as Readings excel file.