

COP 5536 FALL 2013

Programming Project

October 25,2013

Name : Balaji Iyer

UFID : 0443-5000

Email : balajiier@ufl.edu

Table of Contents

1. Project Objective	3
2. Project Environment.....	3
2.1 Language and Development Environment	3
2.2 Compiler.....	3
2.3 Compile Instructions	4
3. Execution	6
4. Expected Result	8
5. Performance Metrics.....	9
6. Actual Results	11
7. Program Structure	11
7.1 Class Structure	11
7.2 Methods:.....	12
7.2 Sequence Diagram	13
8. Conclusion	15

1. Project Objective

The objective is to implement Prim's algorithm to compute the minimum spanning tree (hereafter MST). The algorithm should implement two different data structures:

- Simple Scheme (Using Array)
- Fibonacci Heap Scheme

The performance of both the schemes have to be measured and analyzed. In addition, the program should support following two data input modes:

- Random mode – Accepts vertex count and density as inputs. Compute the MST using both the schemes
- User Input mode – Accepts a file containing undirected graph data with vertices, edges and weight. The output should be displayed on standard output stream

2. Project Environment

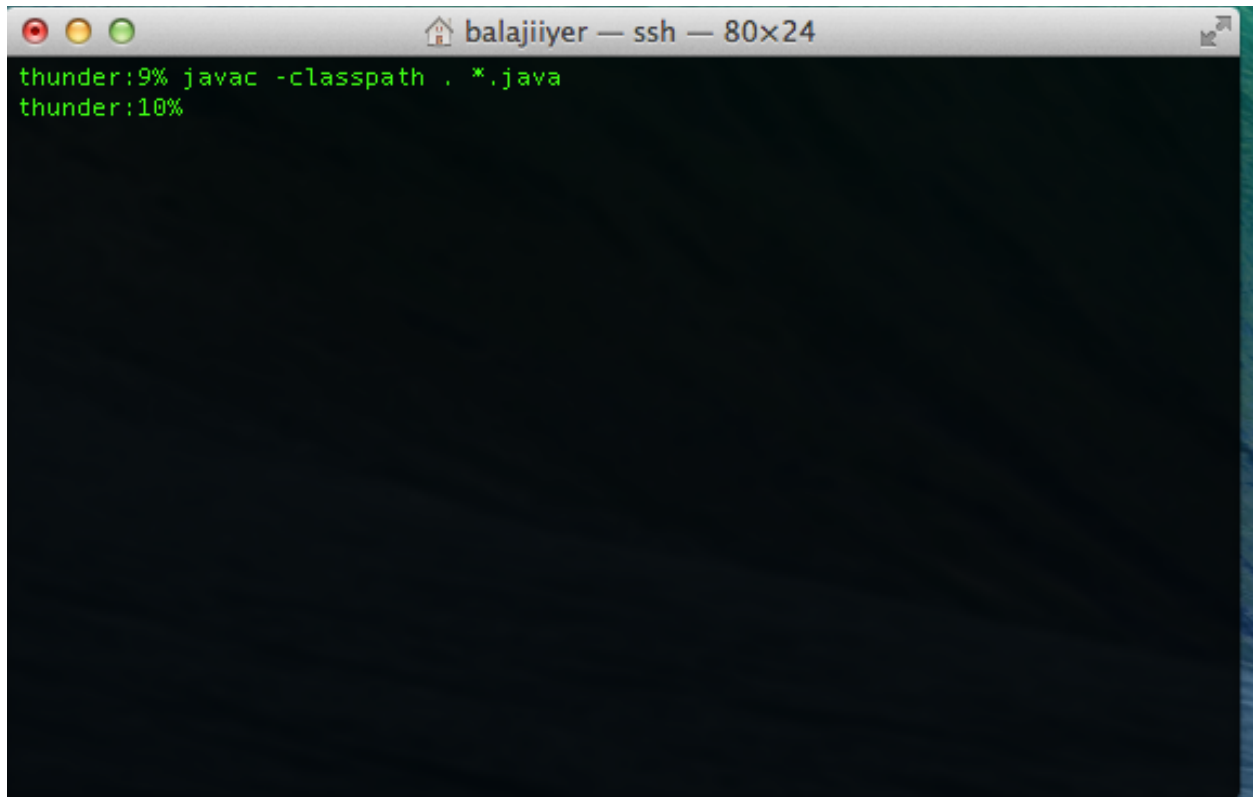
2.1 Language and Development Environment

The program is implemented in *JAVA (v 1.6)*. *Eclipse Juno* is the development environment.

2.2 Compiler

The program has been compiled using : ***javac 1.6.0_65***

2.3 Compile Instructions

A screenshot of a terminal window titled 'balajiiyer — ssh — 80x24'. The terminal shows two lines of text: 'thunder:9% javac -classpath . *.java' and 'thunder:10%'. The background is dark, and the text is green.

```
thunder:9% javac -classpath . *.java
thunder:10%
```

Figure A

To compile the program, execute the following command (refer Figure A):

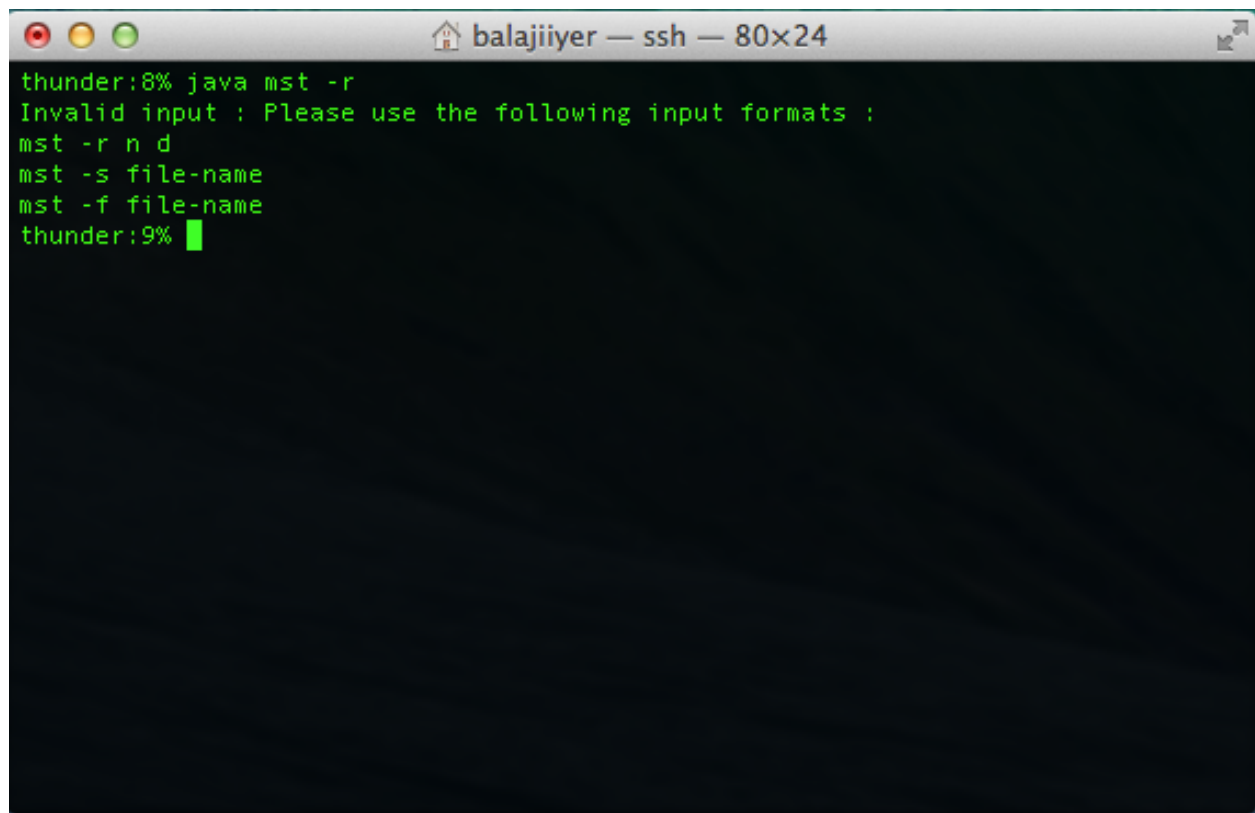
`javac -classpath . *.java`

On successful compilation, all the dependent java files will be compiled and class files will be produced in the same directory.

To verify, execute the following command (refer Figure B)

`java mst -r`

The following output should be displayed:



```
thunder:8% java mst -r
Invalid input : Please use the following input formats :
mst -r n d
mst -s file-name
mst -f file-name
thunder:9% █
```

The image shows a terminal window titled 'balajiiyer — ssh — 80x24'. The prompt is 'thunder:8%'. The user enters 'java mst -r'. The program outputs 'Invalid input : Please use the following input formats :'. It then lists three valid input formats: 'mst -r n d', 'mst -s file-name', and 'mst -f file-name'. The prompt changes to 'thunder:9%' and a green cursor is visible.

Figure B

3. Execution

The program can be executed in the following modes:

- **Random Mode:** Run with graphs generated in random mode.

- *Command line argument: **mst -r n d***

where -r → random mode

-n → number of vertices

-d → density of graph

For e.g.: mst -r 1000 100

- **Expected Output :**

Time for Simple Scheme: X milliseconds

Time for Fibonacci Scheme: X milliseconds

- **User Input mode:** Run with undirected graph data provided by an input file. This mode supports two execution models:

- **Simple Scheme :**

- *Command line argument: **mst -s filename***

where -s → Simple scheme

For e.g.: mst -s mst.txt

- **Expected Output :**

3 2

0 1 5

1 2 8

- **Fibonacci Heap Scheme :**

- *Command line argument: **mst -f filename***

where -f → Fibonacci heap scheme

For e.g.: mst -f mst.txt

- **Expected Output :**

3 2

0 1 5

1 2 8

Note:

- *The filename should contain absolute path. Placing the input file in the same directory of execution is recommended.*
- *In case, the execution fails due to “Out of memory error”, increase the heap size using the following command:*

java -Xms<memsize in mb> -Xmx<memsize in mb> mst -r n d

*For e.g.: **java -Xms512m -Xmx1024m mst -r 5000 100***

4. Expected Result

Prim's algorithm is implemented using a Simple scheme and Fibonacci Heap scheme. In both modes, adjacency list is used to represent the graph. The greedy algorithm finds a minimum spanning tree for an undirected graph. It starts with an arbitrary node and expands on the node with the smallest weight to form a tree. In the end, it includes all the vertices such that the total weight of all edges is minimized.

The simple scheme uses a linked list data structure to compute the minimum spanning tree. The complexity of simple scheme is $O(V^2)$ where V is the number of vertices for a given graph $G=(V,E)$. There are two loops in the scheme. The outer *while* loop visits each vertex adding one vertex at a time to MST. This loop iterates V times. And for each vertex, the inside *for* loop goes through a list of all the edges and finds the smallest weighted edge. So, the inside *for* loop performs V iterations for every *while* loop.

Therefore the number of calculations is proportional to V^2 . Hence, the running time should be $O(V^2)$ for the simple scheme.

Fibonacci heap have asymptotically good run time. The operations insert, decrease key operations run in $O(1)$ amortized time. And removeMin, delete operations run in $O(\log n)$ amortized time. Particularly, the decrease key operation significantly contributes in reducing the run time. Internally, Fibonacci heap is stored as a circular doubly linked list. The complexity of Fibonacci Heap scheme is $O(E + V \log V)$ where the number of edges and V is the number of vertices. The insertions into heap take $O(V)$ time. Then $O(V)$ dequeues are performed since we want only $V-1$ edges. Therefore, dequeues take $O(V \log V)$ time. On each dequeue, all the edges are scanned. All the edges are visited only once, hence it will add to $O(E)$ run time. Therefore, the total run time is $O(E + V \log V)$.

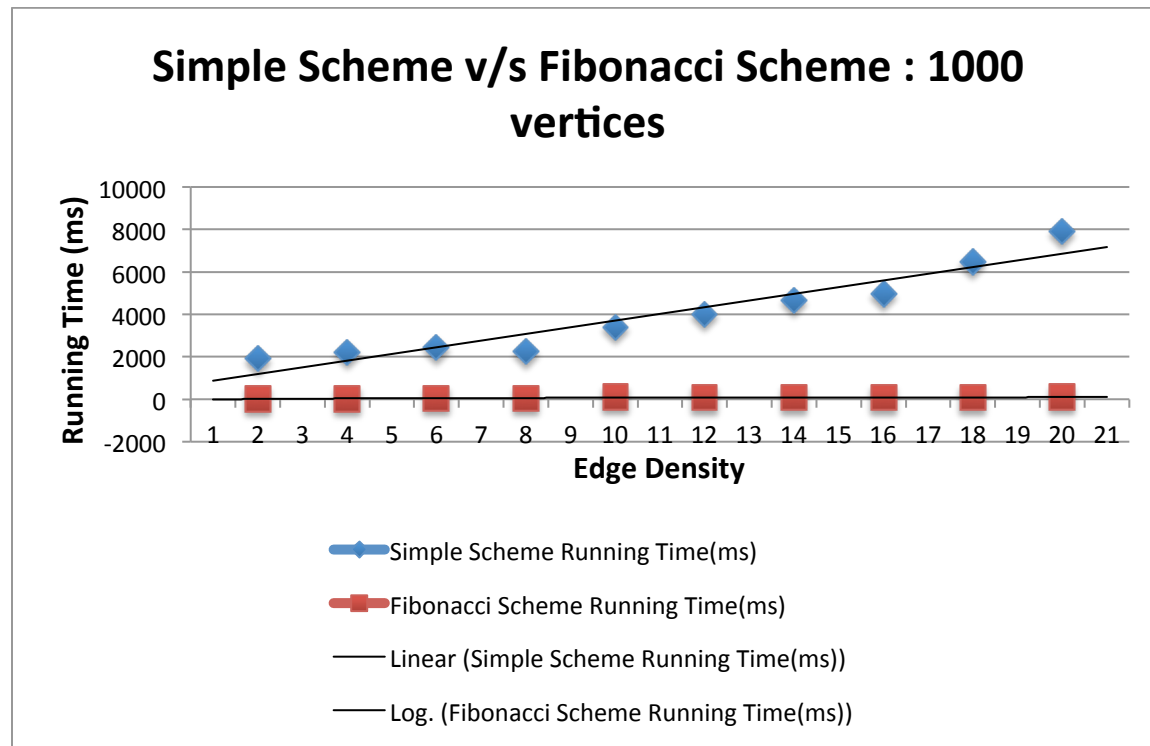
Hence, comparing the run times of both the schemes, the fibonacci heap scheme should perform better than the simple scheme.

5. Performance Metrics

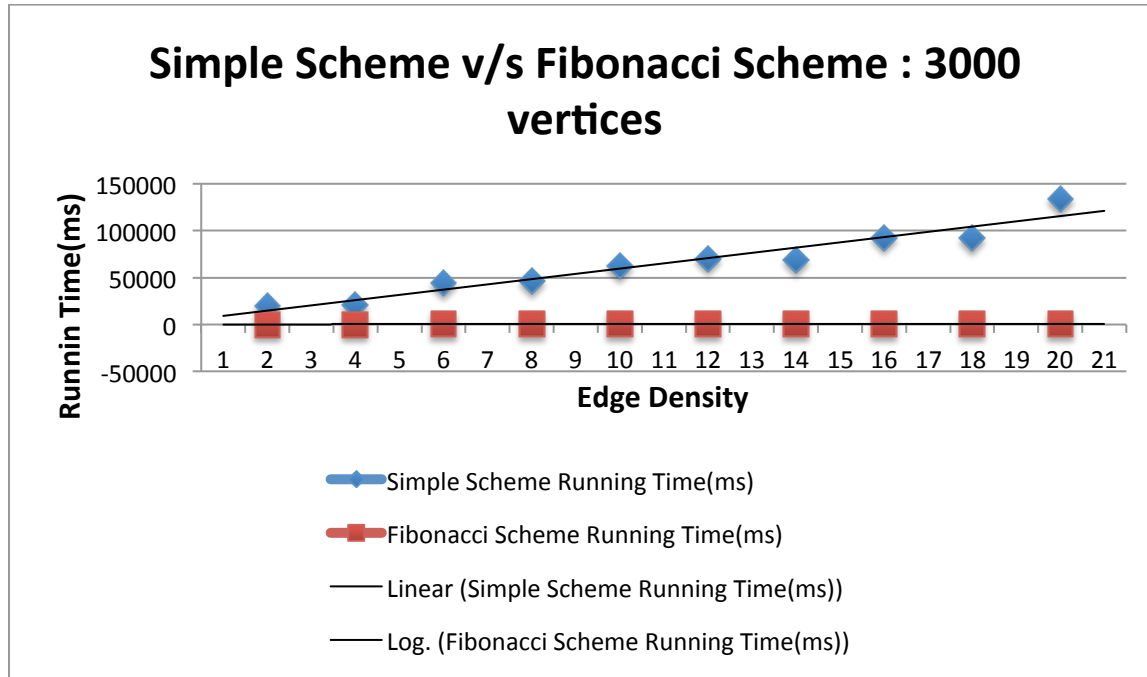
The performance of simple scheme and Fibonacci scheme was measured. Executing the program in random mode has populated the following data for 1000,3000 and 5000 vertices across different densities.The running time is measured in *milliseconds*.

	1000 Nodes		3000 Nodes		5000 Nodes	
Density	Simple scheme	Fibonacci Scheme	Simple scheme	Fibonacci Scheme	Simple scheme	Fibonacci Scheme
10	2000	20	19653	113	62937	245
20	2112	25	20896	225	101200	398
30	2444	52	44067	457	203139	568
40	2260	58	46104	612	204389	827
50	3415	94	62727	523	363901	2610
60	3970	85	70423	908	530129	1230
70	4630	60	67988	843	978774	3568
80	4986	89	85908	638	1127650	1142
90	6449	79	91224	650	1648930	2379
100	7854	104	1108678	910	1883759	21368

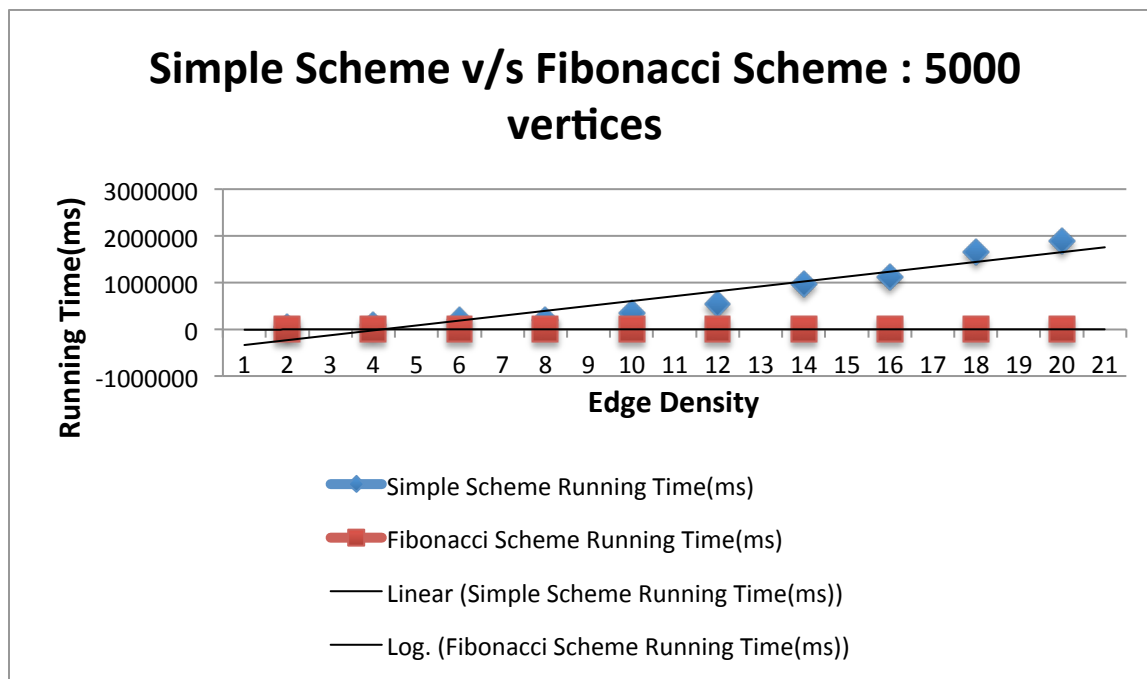
For 1000 vertices:



For 3000 vertices:



For 5000 vertices:



6. Actual Results

As we compare the performance from the results above, we can see that Fibonacci scheme certainly beats the simple scheme in terms of running time. We can see the performance varies as the density changes with the increase in the number of vertices. Fibonacci heap keeps performing better than simple scheme. Also, there are some observations to note. The program requires more memory for dense graphs. Hence, the program execution displays an out of memory exception while generating MST. Increasing the heap size while executing it can mitigate the problem.

7. Program Structure

7.1 Class Structure

Following functions have been implemented:

- mst.java – Initiates the program and computes the mst in random mode and user input mode
 - Inputs : Vertex count and edge density for random mode. File containing graph data for simple and Fibonacci scheme
 - Output: Prints mst to standard output stream
- Graph.java – The Graph class generates an undirected graph for given vertices and edges
 - Inputs : Vertex count and edges
 - Output: Creates a graph
- Edge.java - The Edge class represents an undirected graph with a weighted edge.
 - Inputs : Vertex, edge and weight
 - Output: Creates a node

- Depth First Search.java –Checks whether the graph is connected
 - Inputs : Graph object
 - Output: Boolean connected or not
- FibonacciHeap.java – Holds the structure of a heap
- Input Streamreader.java – Reads a data from a file and returns a scanner object
- PrimMstFibonacciHeapScheme – Implements Prim’s algorithm using Fibonacci Heap
- PrimMstSimpleScheme – Implements Prim’s algorithm using simple scheme(using linkedlist)

7.2 Methods:

These are the basic functions used to compute the mst

Simple Scheme:

- **mstSimpleScheme(Graph graph, Boolean status)**

Takes a graph object and computes mst using linked list implementation. The Boolean status decides the output format of mst.

Fibonacci Heap Scheme:

- **msFibonacciHeapScheme(Graph graph, Boolean status)**

Takes a graph object and computes mst using Fibonacci heap implementation. The Boolean status decides the output format of mst.

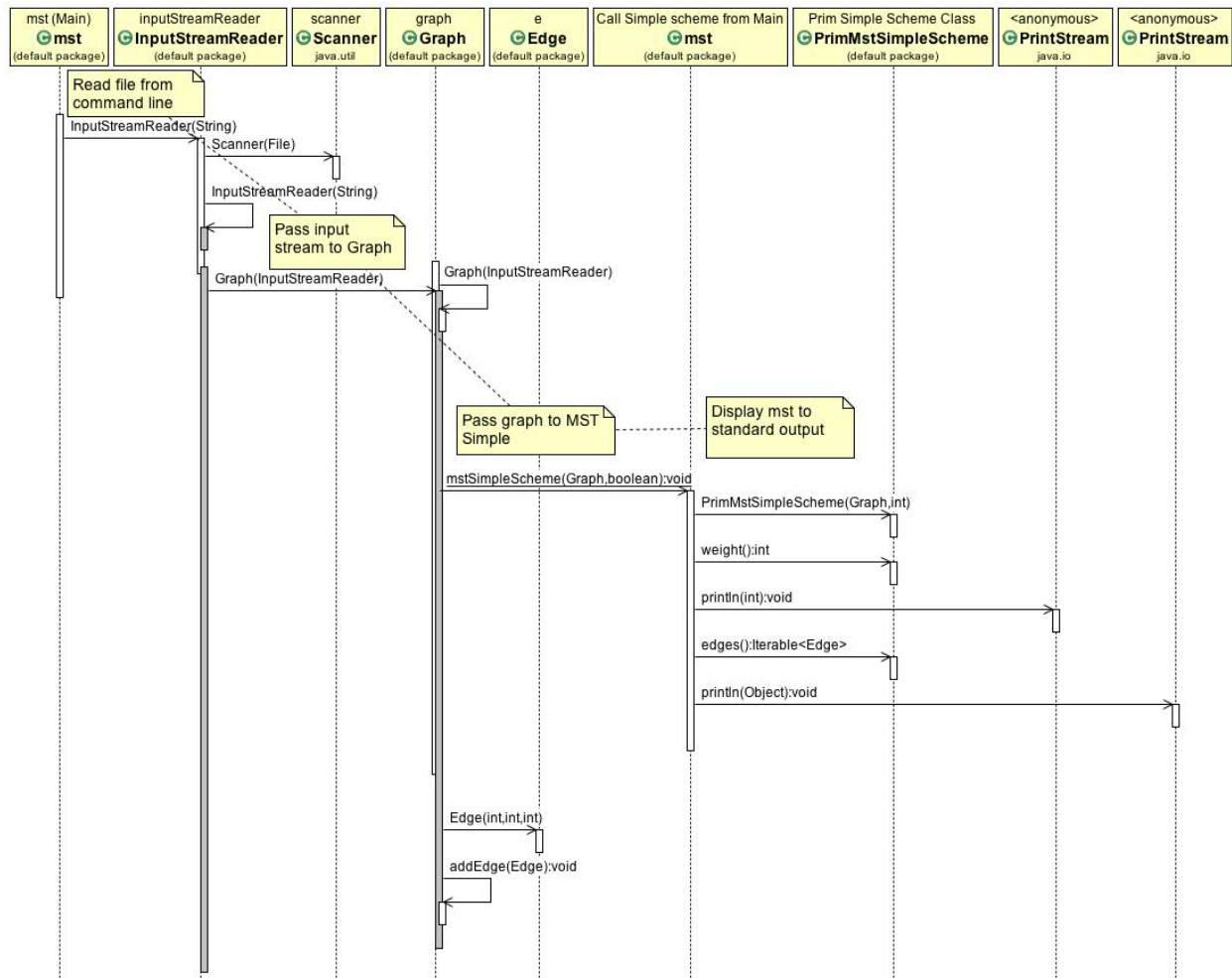
Random Mode:

The random mode uses the above apis to measure the run time for each scheme. This mode takes vertex counts and edge density and generates a graph.

All three modes are executed using command line options.

7.2 Sequence Diagram

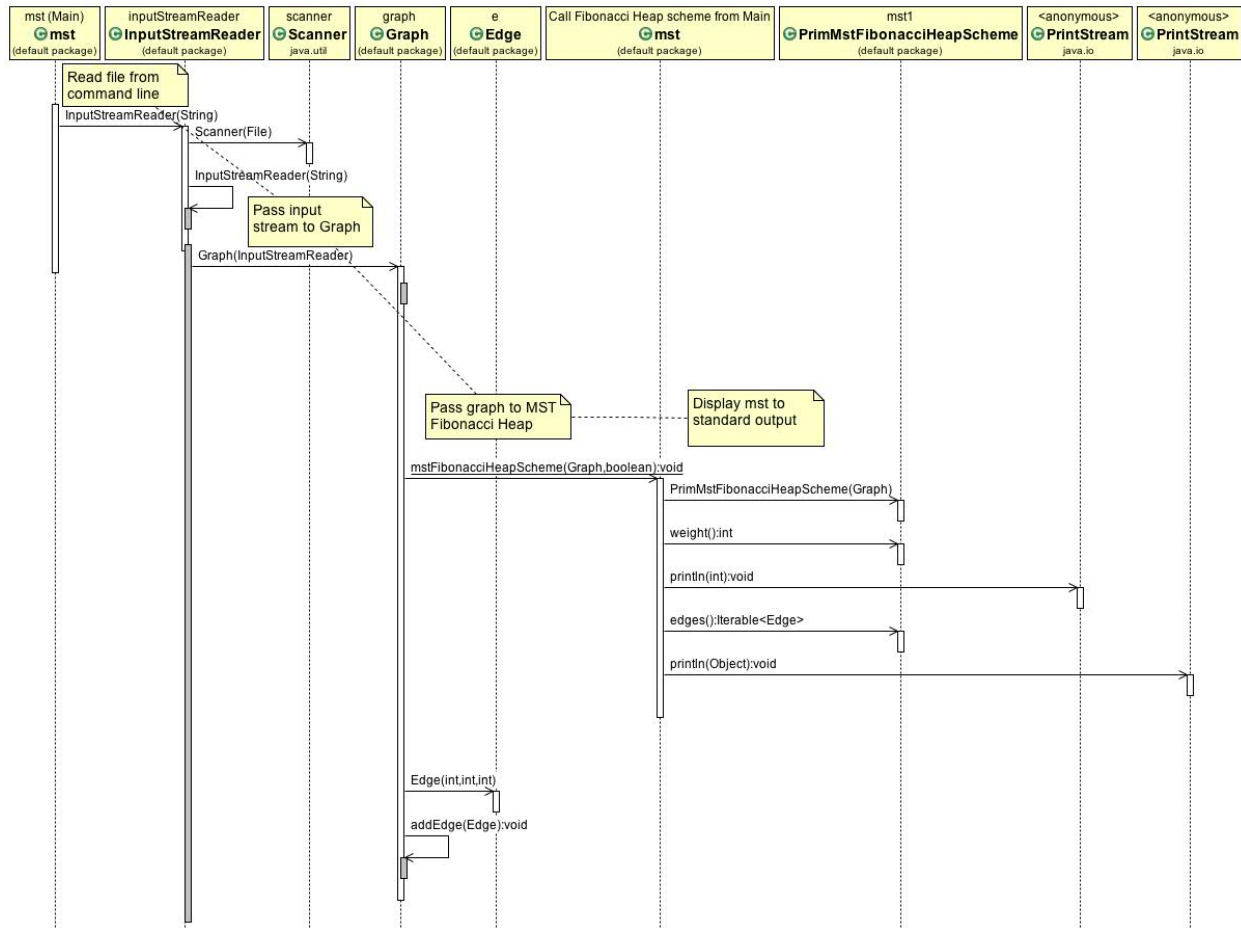
Simple Scheme:



Steps:

- mst main function calls simple scheme method by specifying `-s` option
- InputStreamReader reads the file from command line and passes the object to Graph
- Graph class reads the data from file and creates an undirected graph
- The graph object is passed to PrimMstSimpleScheme class, which computes the mst.

Fibonacci Heap Scheme:



Steps:

- mst main function calls fibonacci scheme method by specifying `-r` option
- Inputstreamreader reads the file from command line and passes the object to Graph
- Graph class reads the data from file and creates an undirected graph
- The graph object is passed to PrimMstFibonacciScheme class, which computes the mst.

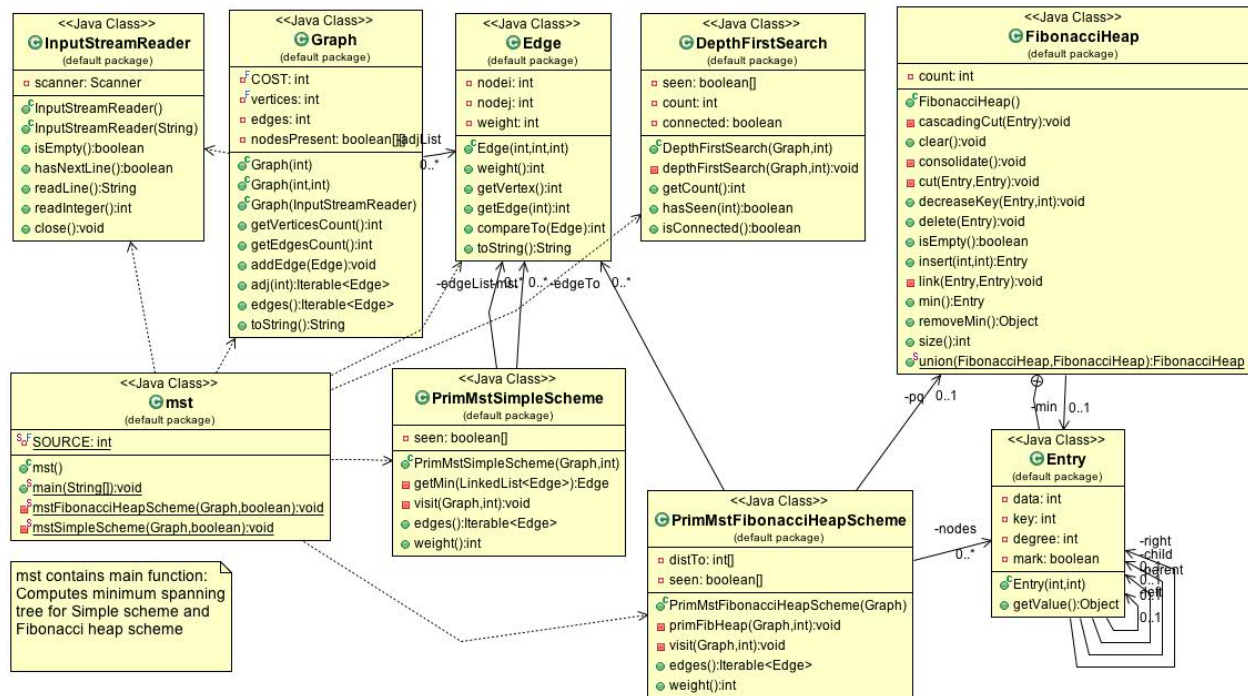
Class Diagram:

Figure above represents various classes that were used to compute Minimum Spanning tree using Prim's algorithm using Simple and Fibonacci heap scheme.

8. Conclusion

It is pretty evident that Fibonacci Heap scheme outperforms array implementation due to performance benefits offered by different operations of Fibonacci Heap scheme.

Observations: For sparse graph, Depth First search consumes lot of time to check if the graph is connected. That is obvious considering the edge density being low.

Memory consumption is more for higher number of vertices. This problem can be evaded by increasing java heap size.