ADVANCED DATA STRUCTURES
PROJECT REPORT
COP 5536

Name : Shreyas Seetharam

UFID: 3815-1424
e-mail:Shreyas.ram@ufl.edu

1) Associated Files

1. **Mst.java :** This class has main function and it initiates the execution. It also has the functions to perform Prim's algorithm

2. **Graph.java:** This class has the adjacency list implementation of a Graph. It also includes methods to generate random graph.

3. **FibonacciHeapNode.java:** This class defines the Single node objects to generate a Fibonacci Heap with. This also keeps track of its neighbours using nodeNex, nodePrev, also has fields for a multidimensional value, a degree, nodeParent,nodeChild to point to parent and child respectively.

4. **FibonacciHeap.java :** The class implements a min Fibonacci Heap Node. Class has various fields such as node degree, value contained in the node, marked value which tells weather any child of node was cut since it became child of its parent. The siblings are stored as a doubly linked list.

2) Compiler Description

Steps to execute the project in LINUX Environment using javac from commandline

1. Untar the .tar file, which contains all the .java files mentioned in the section 1.

2. Navigate to the folder ./MST/src/adsProject/

3. Execute the following command javac –d ../ *.java

4. To the run the application, following command java –cp . ADSProject.mst <– options> in the same directory as .class files.

DETAILED DESCRIPTION -

There are 2 modes of operation. Random Mode and User input mode

RandomMode:

Syntax:

Mst.java –r n d

Where n=number of vertices and d is the density of the graph.

To implement the random mode, a function makeRndGraph() function is called which generates a random graph and calls the performDFS to check if the graph is a completed graph. If it is then the Prim's function PrimAL() which uses simple scheme of adjacency list and PrimFH() which uses Fibonacci Heap implementation is called and the timings for performing this is printed.

User Input Mode:

Syntax:

Mst.java –f <filename>

Mst.java –s<filename>

A check is done to see if it is –f (for FibonacciHeap) or –s(Simple Scheme). Then a Graph constructor is called with the filename as input to generate the graph. Then the corresponding

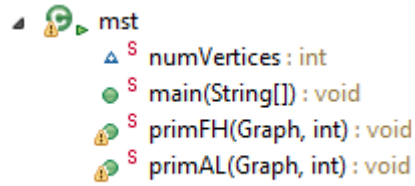
Description of Prim's Algorithm f a graph is empty then we are done

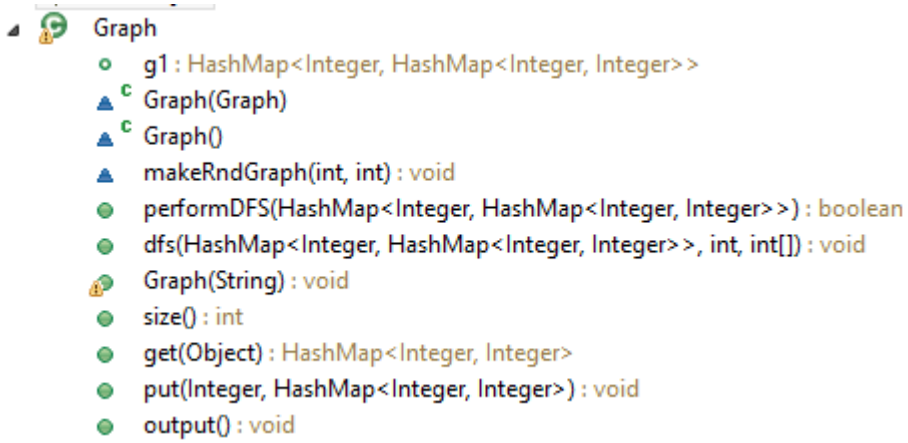
immediately. Thus, we assume otherwise.

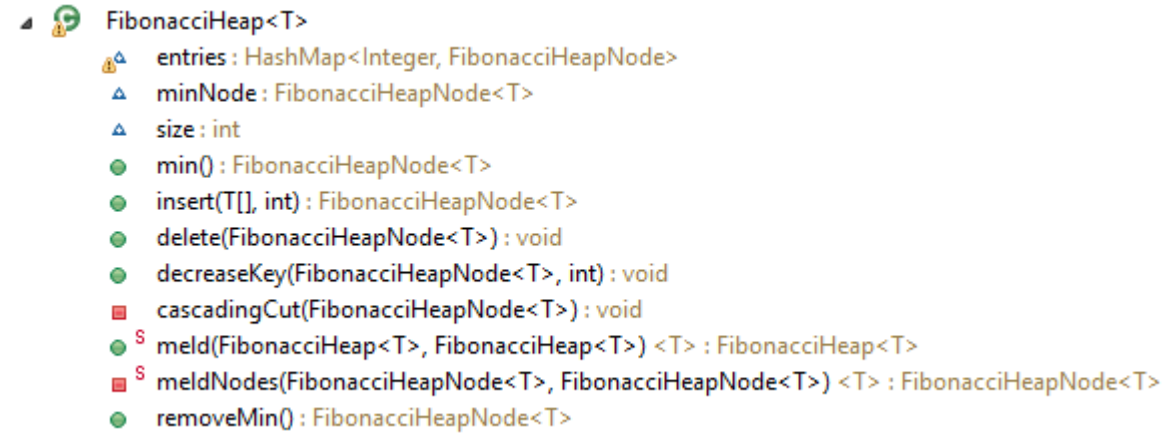
The algorithm starts with a tree consisting of a single vertex, and continuously increases its size one edge at a time, until it spans all vertices.

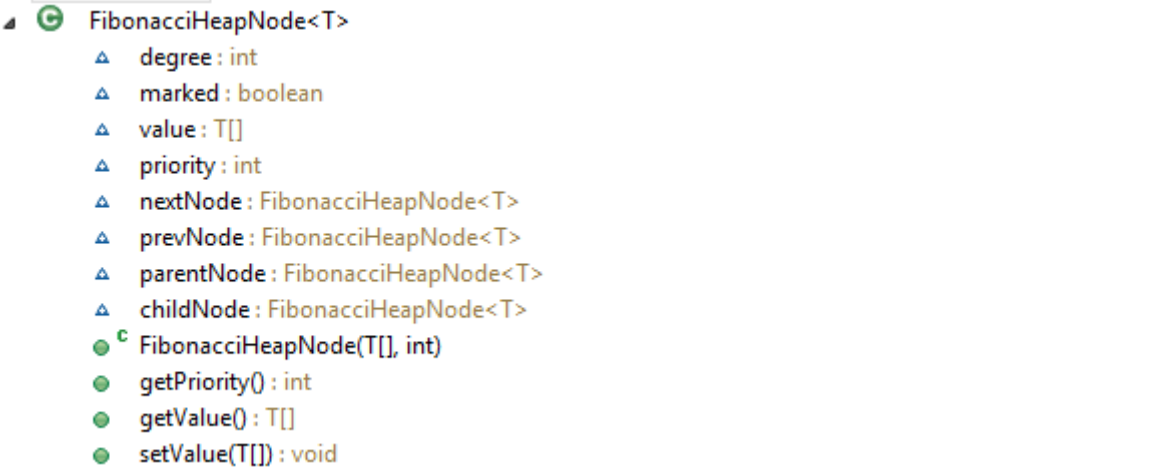
- Input: A non-empty connected weighted graph with vertices V and edges E (the weights can be negative).
- Initialize: $V_{new} = \{x\}$, where x is an arbitrary node (starting point) from V, $E_{new} = \{\}$
  - Repeat until $V_{new} = V$:
- Choose an edge $\{u, v\}$ with minimal weight such that u is in V_{new} and v is not (if there are multiple edges with the same weight, any of them may be picked)
- Add v to V_{new}, and $\{u, v\}$ to E_{new}
- Output: V_{new} and E_{new} describe a minimal spanning tree

PROJECT OUTLINE





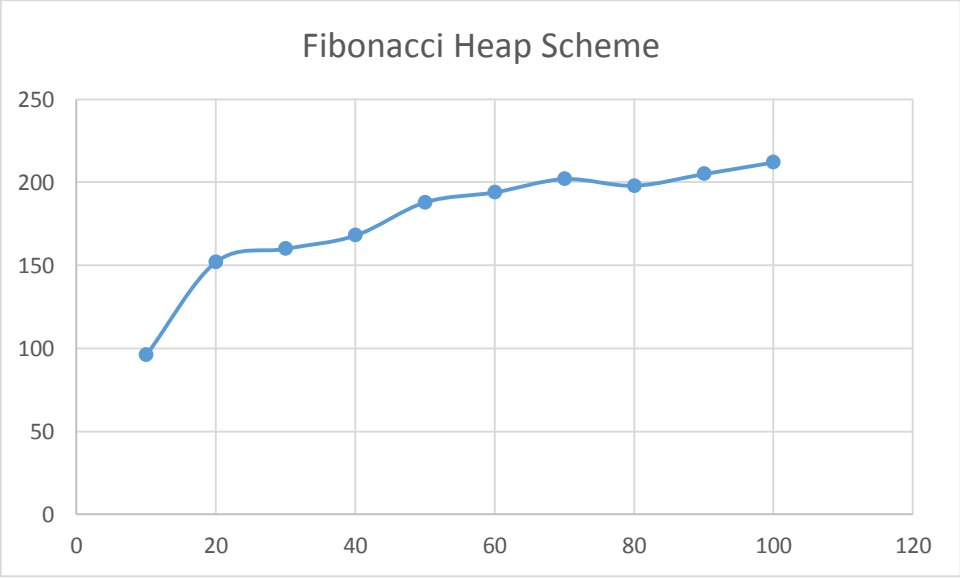

FibonacciHeap<T>
entries : HashMap<Integer, FibonacciHeapNode>
minNode : FibonacciHeapNode<T>
size : int
min() : FibonacciHeapNode<T>
insert(T[], int) : FibonacciHeapNode<T>
delete(FibonacciHeapNode<T>) : void
decreaseKey(FibonacciHeapNode<T>, int) : void
cascadingCut(FibonacciHeapNode<T>) : void
meld(FibonacciHeap<T>, FibonacciHeap<T>) <T> : FibonacciHeap<T>
meldNodes(FibonacciHeapNode<T>, FibonacciHeapNode<T>) <T> : FibonacciHeapNode<T>
removeMin() : FibonacciHeapNode<T>

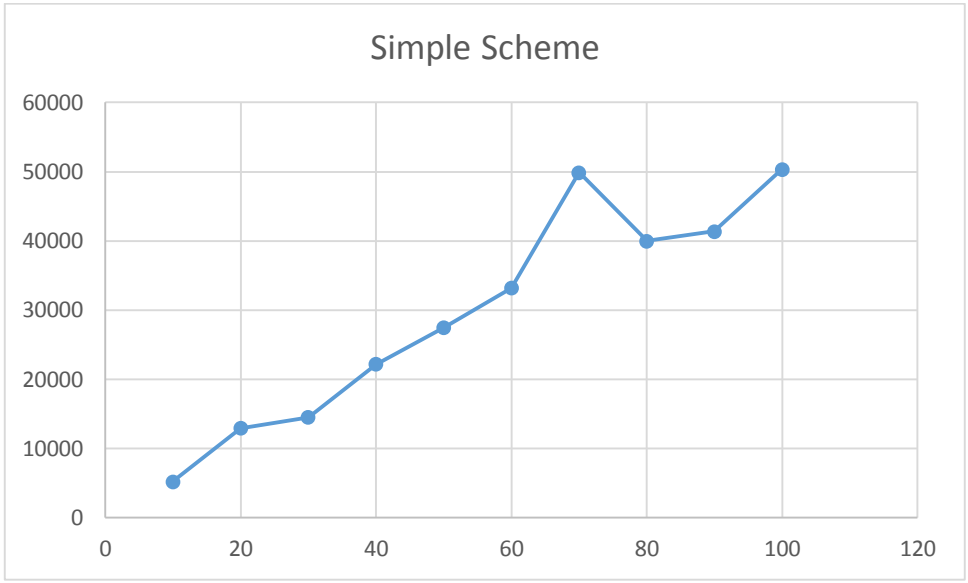

FibonacciHeapNode<T>
degree : int
marked : boolean
value : T[]
priority : int
nextNode : FibonacciHeapNode<T>
prevNode : FibonacciHeapNode<T>
parentNode : FibonacciHeapNode<T>
childNode : FibonacciHeapNode<T>
FibonacciHeapNode(T[], int)
getPriority() : int
getValue() : T[]
setValue(T[]) : void

4. A Summery of Result Comparison

For sparse graphs, Fibonacci scheme gives better results than the simple scheme. For denser graphs as the number of vertices increases, the Fibonacci heap implementation is not affected much, but the simple scheme implementation is greately affected. Some of the results for a number of nodes=1000 are tabulated below.

Density	Fib Heap	Simple Scheme
10	96	5207
20	152	12958
30	160	14538
40	168	22207
50	188	27472
60	194	33178
70	202	49903
80	198	39980
90	205	41358
100	212	50352




Simple Scheme
60000
50000
40000
30000
20000
10000
0
0
20
40
60
80
100
120