COP 5536 FALL 2013
PROGRAMMING PROJECT

Name: Nidhi Aggarwal
UFID: 03149559
Email: Nidhi.aggarwal@ufl.edu

Index

Source code files:

1. **mst.java** – This file has the main method from where our program starts. It checks for:
   - Random mode: -r n d
     This mode runs Prim's implementation using both Simple scheme and Fibonacci heap scheme and calculates time to run both in milliseconds.
   - User Input mode:
     -s file-name: These arguments at the command prompt run Prim's simple scheme
     -f file-name: These arguments at the command prompt run Prim's Fibonacci heap scheme
2. **Vertex.java** – This class defines one vertex object of an array of all vertices each having an adjacency list. It is used to create an adjacency list of a graph.
3. **Neighbour.java** - This class defines each node of each vertex's adjacency list. It stores the following:
   Vertex Number of the second vertex of the edge
   Weight of the edge to the second vertex
   Pointer to the next vertex which is connected to the source vertex in the adjacency list
4. **Graph.java** - This class creates adjacency list for both user input and random generated undirected connected graph
   For User Input graph: This code assumes that the user always gives correct input format and valid data in a "text file". (Costs are always integer type)
   The method **graph_User_Input()** generates adjacency list for user input data
   For random generated graph: The user inputs "number of vertices" and "density" of the graph
   (Costs are always integer type)
   The method **prodGraph()** generates adjacency list of valid connected graph by checking connectivity using Depth first Search
   Depth First Search: DFS() takes a graph and a starting vertex number as input and returns a boolean array showing false at indices which are equal to vertex numbers not reachable by some other vertex
5. **FibHeapNode.java** - This class defines a Fibonacci Heap's node structure. has various fields such as node degree, value contained in the node, childcut value which tells whether any child of node was cut since it became child of its parent, index which indicates where node is placed in the reference array, links left and right which point to some min fibonacci heap roots immediately before and next respectively, child link points to one of the children, parent link points to parent and next link is used when it is required to form the queue of min fibonacci nodes.
6. **FibHeap.java** - This class implements all operations of fibonacci heap such as inserting an element into heap, removing item with minimum value from the heap, arbitrarily removing an element from the heap, decreasing arbitrary node key by changing it to the amount given by user.
7. **PrimsImplemetation.java** - This class implements Prim's algorithm using:
   Simple scheme: in the method **primsBySimple()**

Fibonacci Heap scheme: in the method **primsByFibonacci()**

Complier description:

Platform/OS:

- Windows Edition: Windows 7 Enterprise
- System Type: 64 bit OS
- RAM: 8GB
- Processor: Intel® Core™ i5-2410M CPU @ 2.30 GHz

Compiler:

- Eclipse Compiler for Java (ECJ)
- Javac

How to compile:

Steps to compile using javac:

1. You must have JDK installed on your machine and the PATH environment variable set to where the JDK has your javac application which is in the bin folder generally.
2. Open command prompt: Goto Start → Search cmd
3. Goto the path where your .java files are stored by using cd command. If your .java files are stored in Aggarwal_Nidhi folder, then you should be inside Aggarwal_Nidhi
4. Compile all .java files using the command javac:
   javac *.java
   You will get a .class file for each .java file in the same folder where you have your .java files.

How to run:

Steps to run the project:

1. You must have JRE installed in your machine
2. Open command prompt: Goto Start → Search cmd
3. You must be in the folder where you have your .class files. (Same as the folder where you have .java files in this case)Use cd comand
4. Run the project using the following commands:
   - For random mode:
     java mst –r 'n' 'd'
     where n is the number of vertices and d the density
   - For User input mode by simple scheme:
     java mst –s 'filename'
     where filename is the absolute path of the file and the it's name
   - For User input mode by Fibonacci Heap scheme:
     java mst –f 'filename'
     where filename is the absolute path of the file and the it's name

5. You will see the output on the cmd screen.

Classes, methods and structures:

1. **Mst.java**
   Methods in mst.java:
   public static void main(String[] args) : a) Runs in random mode when user input args[0] is –r: calls prodGraph() of Graph class to which returns random graph(object of Graph class), and then calls primsBySimple() and primsByFibonacci() methods of PrimsImplementaion class by passing the Graph object returned by prodGraph() .

   b) Runs in user input mode when args[0] is either –s or –f. The main method calls graph_User_Input() method which takes filename as inpout and returns an object of Graph class. This Graph object is then input to primsBySimple() iff the argument is –s , or is an input to primsByFibonacci() if the argument is –f.

2. **Vertex.java**
   This class has consists of
   int name;
   Neighbour adjlist;

3. **Neighbour.java**
   int vertexNum;
   int weight;
   Neighbour next;

4. **Graph.java**
   public **Graph graph_User_Input**(String file) throws FileNotFoundException
   This method takes input as a filename and returns adjacency list of the graph formed as an object of Graph class.

   public **Graph prodGraph**(int n, int density)
   This method takes input as number of vertices and the density of the graph and returns a graph's adjacency list which is an object of Graph class if the generated graph is connected.The connectivity is checked by calling the DFS() method.

   public boolean[] **DFS**(Graph chk, int v)

   This method takes an input as a Graph object and a starting vertex and checks whether graph is connected or not by updating a Boolean array everytime it is sure that a vertex is reachable by the vertex v. It then returns this array to the prodGraph() which checks wheter there is any false value in the array. If there is any false value, the prodGraph() method is called again and this continues till we get a connected graph.

5. **public class FibHeapNode**
   This class the following variables:
   int key, val, degree;
   FibHeapNode parent, right, left, child;
   boolean cut;

6. **FibHeap**
   This class implements the functions of Fibonacci Heap required for Prim's algorithm
   public **FibHeapNode** insert(int key, int val)
   This method inserts vertex number and weight of the edge into an object of FibHeapNode and in turn calls private void insert(FibHeapNode x) which inserts this object into the heap.

   public void **decrease**(FibHeapNode tmp, int newKey)
   This method decreases weight/key of the object of FibHeapNode passed to it if its current weight is higher than the passed weight. This method in turn calls inser() and remove() methods to maintain min tre property

   private void **remove**(FibHeapNode x)
   This method removes the FibHeapNode object from the parent and the sibling doubly linked circular list

   private void **join**(FibHeapNode tmp)
   This method combines two min trees generated during delete min

   public int **deleteMin**()
   This method deletes the node with the minimum weight in the heap

7. public class **PrimsImplementaion**
   This class has the following methods:
   public void **primsBySimple**(Graph g)
   This method is called from the main() method of mst class and takes as input an object of Graph generated by graph_User_Input() or prodGraph() methods of Graph class. It calculates the MST and the minimum cost using Prim's algorithm by simple scheme.

   public void **primsByFibonacci**(Graph gr)
   This method is called from the main() method of mst class and takes as input an object of Graph generated by graph_User_Input() or prodGraph() methods of Graph class. It calculates the MST and the minimum cost using Prim's algorithm by deleting the minimum element from the Fibonacci heap.

Expected Result:

1. Simple scheme using arrays to calculate the minimum cost and the MST takes O(v^2) time for both dense an sparse graphs.
2. Running time with fibonacci heap is O(e + v log v) asymptotically which includes the following:
    - Running time is dominated by heap operations
    - DeleteMin takes O(log v) time
    - Decrease key takes O(1) time
    - In the Fibonacci heap we do v DeleteMins and e decreaseKeys = O(v log v + e) where e can be maximum v^1.5
    - So Fibonacci must be good for the whole range of e, i.e. it should be good for both sparse and dense graphs.

Actual Result:

Table of results:

Number of Vertices	Density	Simple scheme	Fibonacci heap
1000	10	16	15
1000	20	15	16
1000	30	32	31
1000	40	48	31
1000	50	48	46
1000	60	63	63
1000	70	64	62
1000	80	79	78
1000	90	79	78
1000	100	94	94
3000	10	112	95
3000	20	79	62
3000	30	173	156
3000	40	128	124
3000	50	89	94
3000	60	169	164
3000	70	157	140
3000	80	110	109
3000	90	204	187
3000	100	157	156
5000	10	173	156
5000	20	282	265
5000	30	282	265

5000	40	282	281
5000	50	360	358
5000	60	391	390
5000	70	269	276
5000	80	180	218
5000	90	280	301
5000	100	300	310

As we can see from the above table Fibonacci is always better or almost the same as that of simple scheme. For sparse graphs Fibonacci is preferred scheme than simple scheme always. For dense graphs the performance of both the schemes is almost the same.

Conclusion

Fibonacci Heap Implementaion of Prim's algorithm is always better for both sparse and dense graphs as it guarantees O(v log v) complexity for sparse graphs and O(v^1.5) complexity for dense graphs.