COP 5536 Fall 2013
Programming Project

Name: **Punyabrata Dhar**, UF ID: **9526-9889**, Email: **brata1987@ufl.edu**

Makefile
The project is an Eclipse project written in Java. If you import the same project in Eclipse attached as *ADSProject.zip*, it will work.

Source Program
The source code is available in the path *~/ADSProject/src/com/ADS/mst/*.
The name of the main class is *mst.java*.

Files Used
1. *mst.java*
2. *Graph.java*
3. *SimpleScheme.java*
4. *FibonacciHeap.java*
5. *FHeapScheme.java*
6. *RandomNumberGenerator.java*

Compiler Used
JDK

Environment
Eclipse Indigo, Juno etc.

Caution
Out of memory error is prevalent in this case. Please update '-Xmx1G' in the Run Configuration (VM arguments section) to increase the Java heap size.

Steps to compile
- Open the project in Eclipse IDE. Select Run → Run Configuration
- In the arguments tab, under program arguments type the command line arguments.
- Run the program.

Structure of The Program

File: Graph.java

- The Graph is built in *Graph.java* file.
- It uses the *Neighbor* and *Vertex* classes to build the graph. The graph has two constructors, one for the *random mode*, the other for the *user input mode*.
- Once the graph is built, the *printGraph()* method prints the graph showing its adjacency list for each vertex.
- The *traverseDFS()* method performs a DFS traversal of the graph and confirms the connectivity of it. In the *user input mode* it prints in the console *The graph is not connected* for an unconnected graph. In the random mode, if its unconnected, it prints the same but loops till we get a connected graph.
- There are two methods namely *findMSTSimple()* and *findMSTFHeap()* that computes the minimum spanning tree (MST) using the Prim's algorithm. The former one uses an array to determine the edges of the MST while the later uses a fibonacci heap to do so.

Function Prototypes

```
/* Method:    Constructor
 * Input:     Integer, Float
 * Output:    Void
 * Description:       This parameterized constructor uses the following things:
 * Integer 'n' to denote the number of vertices; Float 'd' to denote the density
 */
public Graph(int n, float d)

/* Method:    Constructor
 * Input:     String
 * Output:    Nil
 * Description:       This is the parameterized constructor which accepts an input file
 */
public Graph(String fileName)

/* Method:    printGraph()
 * Input:     Nil
 * Output:    Void
 * Description:       This method prints the Graph
 */
public void printGraph()

/* Method:    traverseDFS()
 * Input:     Nil
 * Output:    Void
 * Description:       This method performs the DFS traversal of the Graph
 */

/* Method:    findMSTSimple()
 * Input:     Nil
```

```
 * Output:      Void
 * Description: This method computes the minimum spanning tree of the graph using Simple
Scheme
 * It uses PRIM'S ALGORITHM to find the minimum spanning tree
 */

/* Method:    findMSTFHeap()
 * Input:       Nil
 * Output:      Void
 * Description:         This method computes the minimum spanning tree of the graph using
 * the Fibonacci Heap Scheme
 * It uses PRIM'S ALGORITHM to find the minimum spanning tree
 */
public void findMSTFHeap

/* Method:    getVertices()
 * Input:       Nil
 * Output:      Array of Vertices
 * Description:         This method is a getter for 'vertices'
 */
public Vertex[] getVertices
```

File: SimpleScheme.java

- The Simple Scheme uses the array named *costArray* which is an array of a custom class *EdgeCost*. This class keeps track of the two end-point vertices and the cost of the edge.
- There are methods available to add an edge, invalidate an edge etc.
- This class is used by the *findMSTSimple()* method to compute the MST using the Prim's algorithm.

Function Prototypes

```
/* Method: Constructor
 * Input: Integer
 * Output: Nil
 * Description:         This parameterized constructor takes number of edges as input
 * The same will be the size of the cost-array used in the Simple Scheme
 */
public SimpleScheme

/* Method:    addEdge()
 * Input:       EdgeCost
 * Output:      Void
 * Description:         This method adds the edge to the cost-array
 */
public void addEdge
```

```
/* Method:    extractLocalMin()
 * Input:       Nil
 * Output:     EdgeCost
 * Description:        This method extracts the Edge, with minimum cost, in the cost-array
 */
public EdgeCost extractLocalMin

/* Method:    invalidateEdge()
 * Input:       EdgeCost
 * Output:     Void
 * Description:        This method invalidates an Edge by making 'isUsed = true'
 */
public void invalidateEdge

/* Method:    ignoreEdge()
 * Input:       EdgeCost
 * Output:     Void
 * Description:        This method ignores an Edge by making 'isIgnored = true'
 */
public void ignoreEdge

/* Method:    doesEdgeExist()
 * Input:       EdgeCost
 * Output:     Boolean
 * Description:        This method checks whether an edge already exists in the cost-array
 */
public Boolean doesEdgeExist
```

File: FibonacciHeap.java

- This class implements the Fibonacci Heap.
- A class *FibonacciNode* maintains the node properties of the nodes in each min-tree. It consists of the following attributes. The details description is there in the source file.
    1. key
    2. degree
    3. child
    4. parent
    5. childCut
    6. next
    7. previous
    8. topLevel
    9. graphKey
- The Fibonacci Heap maintains a min item, a Fibonacci Node, which is set as null by the constructor, globally.
- Three important methods of the Fibonacci Heap are implemented here which are used.
- The *insert()* methods inserts a min-tree to the top level Fibonacci Heap and sets the *minItem* pointer if required.

- The *extractMin()* method extracts the minimum element from the Heap. This method in turn uses the *pairwiseCombine()* method to combine the same degree min-trees.
- The *decreaseKey()* method sets a new key to a *FibonacciNode* in the Heap. This method uses the *Cascading Cut* concept if the parent of the node to be decreased key loses more than one child.

Function Prototypes

```
/* Method: Constructor
 * Input: Nil
 * Output: Nil
 * Description:  This non-parameterized constructor initializes the minItem to null
 */
public FibonacciHeap

/* Method:  insert()
 * Input:  FibonacciNode
 * Output:  Void
 * Description:  This method inserts a new node into the Fibonacci Heap
 * The new node is inserted to the top level doubly circular linked list of the heap
 * While inserting, if the top level is empty, we insert it as a first node and make it minItem
 * Else, we check the new minItem in O(1) time and set the minItem accordingly
 */
public void insert

/* Method:  extractMin()
 * Input:  Nil
 * Output:  FibonacciNode
 * Description:  This method extracts the minItem from the tree
 * After this extraction operation, all the subtrees of this minItem, and all the other
 * top level Min-Trees are processed with the following operation.
 * The roots of all the above Min-Trees are broken from their doubly linked list chain
 * All these roots are pushed into an ArrayList of FibonacciNode-s
 * The ArrayList is sent to the method pairwiseCombine()
 * The minItem is returned
 */
public FibonacciNode extractMin

/* Method:  pairwiseCombine()
 * Input:  ArrayList<FibonacciNode>
 * Output:  Void
 * Description:  This method performs the pairwise-combine operation for the
minTrees
 * It maintains a Tree Table which keeps track of the degree of the tree
 * The method traverses through the minTrees one by one
 * It marks the degree of the current minTree to 'true' in the designated cell
 * that points to that minTree
```

```
 * If the mark was already 'true',
 * the method combines the two trees by calling combineMintrees()
 * When all the minTrees are checked for combination, we connect the final minTrees
 * using doubly circular linked list and set the minItem pointer
 */
private void pairwiseCombine

/* Method:    combineMintrees()
 * Input:       FibonacciNode, FibonacciNode
 * Output:     FibonacciNode
 * Description:         This method combines two Min-Trees
 * It checks for the key of minTree1 and minTree2, whichever has the bigger key,
 * that becomes the child of the other
 * The final Min-Tree has got its degree increased by 1
 */
private FibonacciNode combineMintrees

/* Method:    setPointers()
 * Input:       FibonacciNode, FibonacciNode
 * Output:     FibonacciNode
 * Description:         This method makes t1 the child of t2
 * It also sets the parent, child, next and previous pointers accordingly
 */
private FibonacciNode setPointers

/* Method:    decreaseKey()
 * Input:       FibonacciNode, int
 * Output:     Void
 * Description:         This method sets a new key 'newKey' to a given FibonacciNode
 * It performs Cascading Cut if the Min-Tree property is violated
 * It sets new minItem pointer if the given FibonacciNode is a root node
 */
public void decreaseKey

/* Method:    printFibonacciHeap()
 * Input:       Nil
 * Output:     Void
 * Description:         This method prints the entire Fibonacci Heap
 */
public void printFibonacciHeap

/* Method:    printHeap()
 * Input:       FibonacciNode
 * Output:     Void
 * Description:         This method prints the Heap given a start node
 */
```

private void printHeap

/ Method: printList()*
** Input: FibonacciNode*
** Output: Void*
** Description: This method prints the circular doubly linked list at every level*
**/*
private void printList

File: FHeapScheme.java

- This class implements the algorithm that is used to access the Fibonacci Heap to compute the minimum spanning tree. There are two parts of this algorithm.
    1. Initialize
    2. Loop
- In the *initialize* step, which is done in the constructor, all the nodes are inserted one by one in the Fibonacci Heap. The key and edge attributes of a *FibonacciEdge*, which is another class, are set to their required values. A detailed description is given in the source file.
- In the Loop step it runs the actual algorithm that uses the *extractMin*() and *decreaseKey*() function of the Fibonacci Heap. This is done in the method *computeMST*().

Function Prototypes

/ Method: Constructor*
** Input: Graph*
** Output: Nil*
** Description: This parameterized constructor takes the Graph as input and initializes it*
**/*
public FHeapScheme

/ Method: computeMST()*
** Input: Nil*
** Output: HashMap<Integer, FibonacciEdge>*
** Description: This method computes the minimum spanning tree using the Fibonacci Heap*
**/*
public HashMap<Integer, FibonacciEdge> computeMST

File: RandomNumberGenerator.java

This class implements the random number generator using the method *getRandomNumber*().

Function Prototypes

/ Method: getRandomNumber()*
** Input: Integer*
** Output: Integer*
** Description: This method generates an integer random number between 0 and*

'*seek*'

**/*

public static int getRandomNumber

File: mst.java

This class is the main class. Following are the run modes.

User Input mode

The command line arguments for this mode is given as:

-s *Graph.txt* for the Simple Scheme

-f *Graph2.txt* for the Fibonacci Heap Scheme

There are some sample files in the path *~/ADSProject*. Some of these are connected graphs, some are not.

Random mode

The command line arguments for this mode is given as:

-r 1000 50 for both the Simple and the Fibonacci Scheme

The results, given a set of inputs, are displayed in the next section. In the command line argument -r is followed by the number of vertices *n* and the edge density *d*. The number of edges e will be found by the formula e = *Math.ceil*(((n*(n-1)/2)*d)/100).

Function Prototypes

This is the main program. It accepts two input modes.

/* Method: main()

* Input: CommandLine Arguments

* Output: Void

* Description: This method is the main method()

*/

Performance Measurement

For the Simple Scheme, the expected time to find the MST is $O(n^2)$ where n is the number of vertices of the graph.
For the Fibonacci Heap Scheme, the runnning time varies with $O(n log n + e)$ where n is the number of vertices, and e is the number of edges, of the graph.

For dense graph where e is $O(n^2)$ the running time for the two schemes should be same. On the other hand, for a sparse graph where e is O(n), Fibonacci Heap Scheme produces better runnning time. However, improvement in the running time using the Fibonacci Heap is rarely faster than the naive implementation (Simple Scheme) unless the graph is extremely large and dense.

The expectation is that the Fibonacci Heap Scheme will be faster than the Simple Scheme.

Result Comparison (Time in sec)

n = 1000

Edge density (d%)	Simple Scheme	Fibonacci Scheme
10	0.299	0.29
20	0.302	0.3
30	0.309	0.28
40	0.301	0.29
50	0.3	0.3
60	0.298	0.28
70	0.298	0.29
80	0.299	0.27
90	0.3	0.34
100	0.305	0.28

n = 3000

Edge density (d%)	Simple Scheme	Fibonacci Scheme
10	9.829	1.56
20	10.021	1.69
30	12.595	1.74
40	10.832	1.70
50	10.996	1.58
60	10.466	1.54
70	10.869	1.62
80	Java Out of Memory Error	Java Out of Memory Error
90	Java Out of Memory Error	2.15
100	Java Out of Memory Error	Java Out of Memory Error

Result Comparison (Time in sec) (Continued)

n = 5000

Edge density (d%)	Simple Scheme	Fibonacci Scheme
10	755.113	208.01
20	900.023	233.023
30	Java Out of Memory Error	356.543
40	Java Out of Memory Error	Java Out of Memory Error
50	Java Out of Memory Error	Java Out of Memory Error
60	Java Out of Memory Error	Java Out of Memory Error
70	Java Out of Memory Error	Java Out of Memory Error
80	Java Out of Memory Error	Java Out of Memory Error
90	Java Out of Memory Error	Java Out of Memory Error
100	Java Out of Memory Error	Java Out of Memory Error

From the results it is clear that, Out of Memory Error is a very big issue here. However, Fibonacci Heap Scheme outperforms the Simple Scheme in all the above cases. For the dense graph, because of the Out of Memory Error, the expectation cannot be verified.