Fibonacci Heap, Min heap and Simple Array implementation for Prim's Algorithm

Ying She
University of Florida
Gainesville, FL 32611, USA
yshe@ufl.edu

October 25, 2012

Contents

1 File Structure and Method to Compile

All my source files are in the directory "ying", the structure of my source files is as follows:

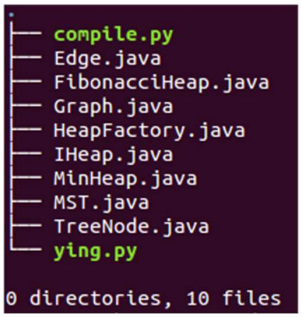

Figure 1: File Structure

Even all my files are in the same directory, they are actually organized into different "package".. There are several ways you can use to run my program. Either import my source files into a new project (use any IDE you like, such as Eclipse or NetBeans), then use IDE to feed command options or as the my files indicated, run my program in a terminal in a operating system which already has JDK and Python installed.

1. **How to compile:** "python compile.py"
2. **How to run:** "python ying.py" or "java com/yingshe/MST" (then followed the instructions)

I have run all these commands in our "thunder.cise.ufl.edu" server and get correct responses, so you should execute them without any difficulty. If encounter anything wrong, contact me.
In fact, I have make "compile.py" and "ying.py" executable using Linux command *chmod*, if you are using Linux or Mac, just execute "./compile.py" and "./ying.py".

2 Sytem Flow Chart

The flow chart of this program is as the following figure:

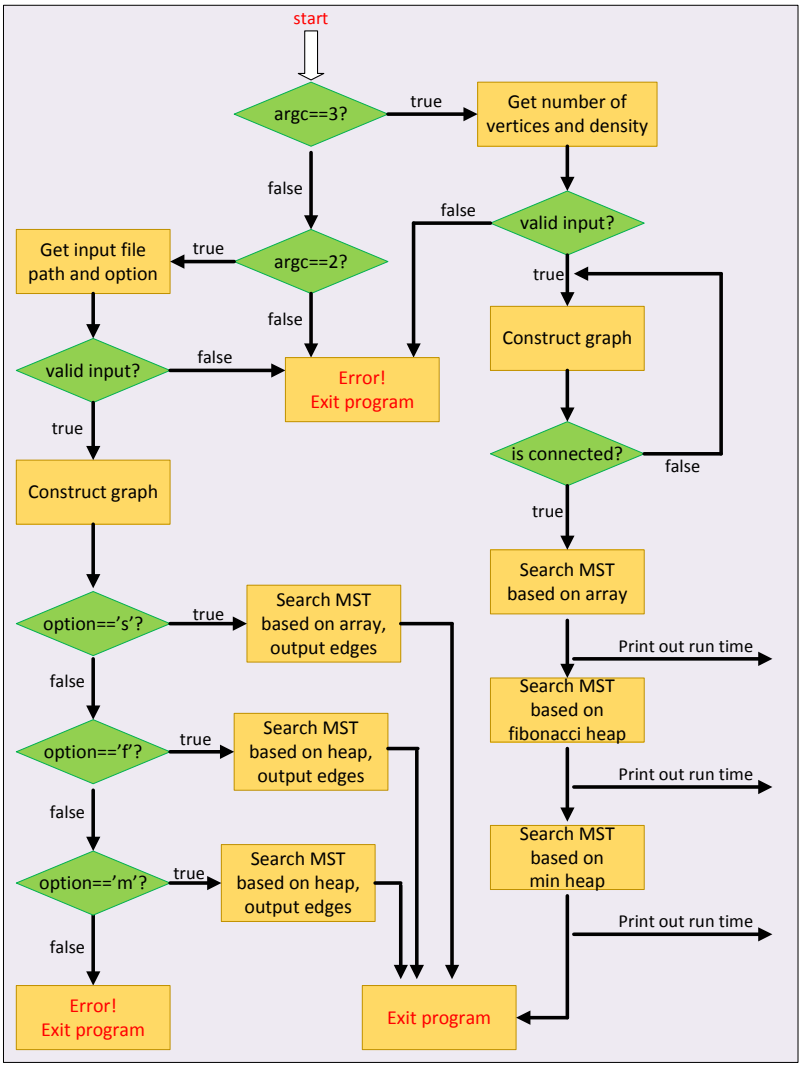

Figure 2: flow chart

3 Class Structure and Definitions

I use Factory Pattern to design my program.

1. **IHeap** is a public interface which have fundamental functionalities, both concrete class **Fibonacci-Heap** and **MinHeap** will implements it. **HeapFactory** will return corresponding instance instructed by user.

2. **TreeNode** is the tree node of heap. (*in fact, min heap's tree node has different fields from Fibonacci heap tree node, so in this TreeNode implementation, I include all fields from both heaps*).

3. **Graph** has adjacency list to represent a undirected graph.

3.1 IHeap

```java
package com.yingshe.heap;

public interface IHeap
{
        abstract public void insert(TreeNode singleNode);
        abstract public TreeNode getMin();
        abstract public boolean removeMin();
        abstract public boolean decreaseKey(TreeNode theNode, int _newKey);
        abstract public boolean isEmpty();
}
```

3.2 HeapFactory

```java
package com.yingshe.heap.factory;

import com.yingshe.heap.*;
import com.yingshe.heap.fiheap.*;
import com.yingshe.heap.minheap.*;

public class HeapFactory{
        public IHeap getHeap(String type)
        {
                //default return Fibonacci heap
                if(type == null) return new FibonacciHeap();
                else if(type.equalsIgnoreCase("Fibonacci"))
                        return new FibonacciHeap();
                else if(type.equalsIgnoreCase("min"))
                        return new MinHeap();
                else
                        return new FibonacciHeap();
        }
}
```

3.3 TreeNode

```java
package com.yingshe.heap;

public class TreeNode{
                public TreeNode()
                {
                        degree = 0;
                        child = null;
                        leftSibling = rightSibling = this;
                        childCut = false;
                        leftChild = rightChild = parent = null;
                        numInLeftSubtree = 0;
                        numInRightSubtree = 0;
                }
                public TreeNode(int key){
                        this();
                        this.key = key;
                }
                public TreeNode(int key, int identity)
                {
```

```
                        this();
                        this.key = key;
                        this.identity = identity;
                }
                public TreeNode(int key, int identity, int neighbor){
                        this(key, identity);
                        this.neighbor = neighbor;
                }
                //getter and setter for "key", "identity" and "neighbor"
                public int getKey(){return key;}
                public void setKey(int _newKey){key = _newKey;}
                public int getIdentity(){return identity;}
                public void setIdentity(int identity){this.identity = identity;}
                public int getNeighbor(){return neighbor;}
                public void setNeighbor(int neighbor){this.neighbor = neighbor;}

                //below member variables are owned by Fibonacci heap
                public int degree;
                public TreeNode child;
                public TreeNode leftSibling;
                public TreeNode rightSibling;
                public boolean childCut;

                //below four member variables are shared by min heap and Fibonacci heap
                public int identity;
                public int key;
                public int neighbor;
                public TreeNode parent;

                //below member variables are owned by min heap
                public TreeNode leftChild;
                public TreeNode rightChild;
                public int numInLeftSubtree;
                public int numInRightSubtree;
}
```

3.4 Graph Representation

Just as the project specification describes, we need to use "adjacency lists" to represent the *undirected graph* (if use "adjacency matrix", the time to search the matrix will dominate the time used by Fibonacci heap, thus give us illusion that Fibonacci is not as good as expected). My "class Graph" is acted as a "undirected graph" which has some functions:

```
package com.yingshe.graph;

import java.util.*;
import java.lang.Math;
import java.io.*;

public class Graph{
        private ArrayList<ArrayList<Edge>> udGraph;
        private boolean connected;
        private int numberOfVertices;
        private int numberOfEdges;

        public Graph()
        {
```

```
15 |                 numberOfVertices = numberOfEdges = 0;
16 |                 connected = false;
17 |                 udGraph = new ArrayList<ArrayList<Edge>>();
18 |         }
19 |
20 |         public Graph(Graph g)
21 |         {
22 |                 numberOfVertices = g.numberOfVertices;
23 |                 numberOfEdges = g.numberOfEdges;
24 |                 connected = g.connected;
25 |                 udGraph = new ArrayList<ArrayList<Edge>>(g.udGraph);
26 |         }
27 |
28 |         public boolean isConnected(){return connected;}
29 |
30 |         public boolean checkConnectedness()
31 |         {
32 | /*
33 |                 if(udGraph.size() == 0 || udGraph.size() != numberOfVertices)
34 |                 {connected = false; return false;}
35 |
36 |                 ArrayList<Boolean> con = new ArrayList<Boolean>();
37 |                 for(int i = 0; i < numberOfVertices; i++)
38 |                         con.add(false);
39 |
40 |                 con.set(0, true);
41 |                 int[] count = new int[1];
42 |                 count[0] = 1;
43 |
44 |                 dfs(0, con, count);
45 |                 connected = (count[0]==numberOfVertices?true:false);
46 | */
47 |                 connected = bfs();
48 |                 return connected;
49 |         }
50 |
51 |         public boolean buildRandomGraph(int vertices, int edges)
52 |         {
53 |                 if(vertices <= 0 || edges < vertices-1 || edges > vertices*(vertices-1)/2)
54 |                         return false;
55 |                 this.numberOfVertices = vertices;
56 |                 this.numberOfEdges = edges;
57 |                 this.udGraph = new ArrayList<ArrayList<Edge>>();
58 |                 for(int i = 0; i < numberOfVertices; i++)
59 |                         this.udGraph.add(new ArrayList<Edge>());
60 |
61 |                 boolean[][] matrix = new boolean[vertices][vertices];
62 |
63 |                 int e = 0;
64 |                 while(e < edges)
65 |                 {
66 |                         //Math.random() generate real number in [0.0, 1.0)
67 |                         int v1 = (int)(Math.random()*vertices);
68 |                         int v2 = (int)(Math.random()*vertices);
69 |                         int distance = (int)(Math.random()*1000)+1;
70 |                         if(v1 != v2)
71 |                         {
72 |                                 if(matrix[v1][v2] == false)
73 |                                 {
```

```
                                matrix[v1][v2] = matrix[v2][v1] = true;
                                e++;
                                Edge p1 = new Edge(v2, distance);
                                udGraph.get(v1).add(p1);
                                Edge p2 = new Edge(v1, distance);
                                udGraph.get(v2).add(p2);
                        }
                }
        }
        return true;
}

public boolean buildGraphFromFile(String filePath)
{
        try{
                Scanner s = new Scanner(new File(filePath));
                if(s.hasNextInt()) numberOfVertices = s.nextInt();
                if(s.hasNextInt()) numberOfEdges = s.nextInt();
                if(numberOfVertices <= 0 || numberOfEdges < numberOfVertices -1)
                        return false;

                udGraph = new ArrayList<ArrayList<Edge>> ();
                for(int i = 0; i < numberOfVertices; i++)
                        udGraph.add(new ArrayList<Edge>());

                int e = 0;
                while(e < numberOfEdges)
                {
                        int v1, v2, distance;
                        v1 = s.nextInt();
                        v2 = s.nextInt();
                        distance = s.nextInt();
                        Edge p1 = new Edge(v2, distance);
                        udGraph.get(v1).add(p1);
                        Edge p2 = new Edge(v1, distance);
                        udGraph.get(v2).add(p2);
                        e++;
                }
                s.close();
        }
        catch(Exception e)
        {
                System.out.println("In class Graph, function buildGraphFromFile,
                    error in opening file: " + filePath);
        }
        return true;
}


public ArrayList<Edge> getNeighbors(int index)
{
        if(index >= numberOfVertices) return null;
        return udGraph.get(index);
}

public int getNumberOfVertices()
{ return numberOfVertices; }

public void setNumberOfVertices(int vertices)
```

```
132         { numberOfVertices = vertices; }
133 
134         public int getNumberOfEdges()
135         { return numberOfEdges; }
136 
137         public void setNumberOfEdges(int edges)
138         { numberOfEdges = edges; }
139 
140         private void dfs(int index, ArrayList<Boolean> con, int[] count)
141         {
142                 if(count[0] == numberOfVertices) return;
143                 for(int i = 0; i < udGraph.get(index).size(); i++)
144                 {
145                         int destination = udGraph.get(index).get(i).destination;
146                         if(con.get(destination) == false)
147                         {
148                                 con.set(destination, true);
149                                 count[0]++;
150                                 dfs(destination, con, count);
151                         }
152                 }
153         }
154 
155         private boolean bfs()
156         {
157                 boolean[] visited = new boolean[numberOfVertices];
158                 Queue<Integer> tool = new LinkedList<Integer>();
159                 int i = 0;
160                 while(i < numberOfVertices && udGraph.get(i).size() == 0)
161                         i++;
162 
163                 tool.offer(i);
164                 visited[i] = true;
165                 int count = 1;
166 
167                 while(!tool.isEmpty())
168                 {
169                         int tmp = tool.poll();
170                         ArrayList<Edge> neighbors = udGraph.get(tmp);
171                         for(Edge e : neighbors)
172                         {
173                                 if(visited[e.destination] == false)
174                                 {
175                                         tool.offer(e.destination);
176                                         visited[e.destination] = true;
177                                         count++;
178                                 }
179                         }
180                 }
181                 return count==numberOfVertices;
182         }
183 }
```

4 Fibonacci Heap Details

I design all functions instructed by the teacher, in my **FibonacciHeap** implementation, I write several private functions in addition to those inherit from **IHeap**:

```
        //for below function, please draw figures to get the idea why I write as this.
        //parameter "first" and "second" are guaranteed not to be "null" (caller
        //guaranteed), but for safety, I check it again inside the function.
        // Please be aware, both "first" and "second" are pointers to doubly circular linked
        // list: a doubly circular linked list could have one node or more nodes
        private void mergeTwoCircularLinkedList(TreeNode first, TreeNode second)
        {
                .......
        }

        //this private function pull out "current" from its own doubly circular linked list,
        // and then insert "current" into  top level doubly circular linked list. Please be
        // aware, "current" can be only one single node or the root of a "min tree"
        void extractAndInsert(TreeNode current)
        {
                ......
        }

        //this private function is written to do cascading cut
        //param TreeNode "current" is the node we are going to do cascading cut
        //return TreeNode is the node which stop us
        // In fact "current" are guaranteed to have parent and have "childCut" as "true"
        // (guaranteed by caller,  but I check this again inside the function.
        private TreeNode cascadingCut(TreeNode current)
        {
                ...
        }

        //define pairwiseCombine as private function since this function only used
        //by other functions, users shouldn't know this function is working.
        //This function would modify "minimumPointer" and "topLevelDegree"
        //In order to do pairwise combine, we need a "tree table" just as the
        // slides indicate
        private void pairwiseCombine(TreeNode temporaryPointer)
        {
                ....
        }
```

The only important thing in addition to **pairwise combine** and **cascading cut** is be careful when merge two doubly circular linked lists:

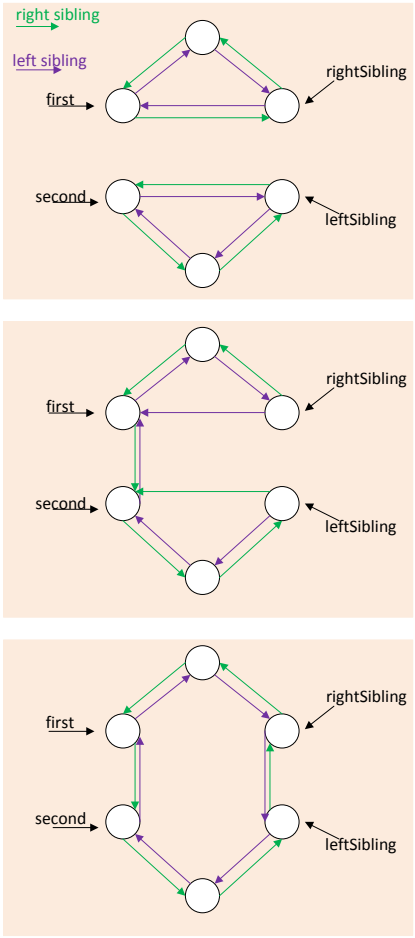

Figure 3: Steps show how to marge two doubly circular linked list

5 Min Heap Details

For min heap, I didn't do it as we learned in class:
In class:

1. insert: first put the node at last, then adjust bottom-up;
2. get min: as usual;
3. remove min: select the last leaf node, put it in the root, then adjust top-down;
4. decrease key: as usual, adjust upward;

My implementation:

1. insert: insert top-down, do comparison along the path downward;
2. get min: as usual;
3. remove min: select one of root's children to take the root's position, then do this recursively until get to last level;

4. decrease key: as usual, adjust upward;

You can see the difference between my implementation and what we was taught in class, I do this way since it's very hard and inefficient to get the *last node* or *last leaf node*. But in order to keep the min tree as balanced as possible, I defined **numInLeftSubtree** and **numInRightSubtree** to instruct the **insert** action: the goal is try to keep $numInLeftSubtree == numInRightSubtree$ for each node:

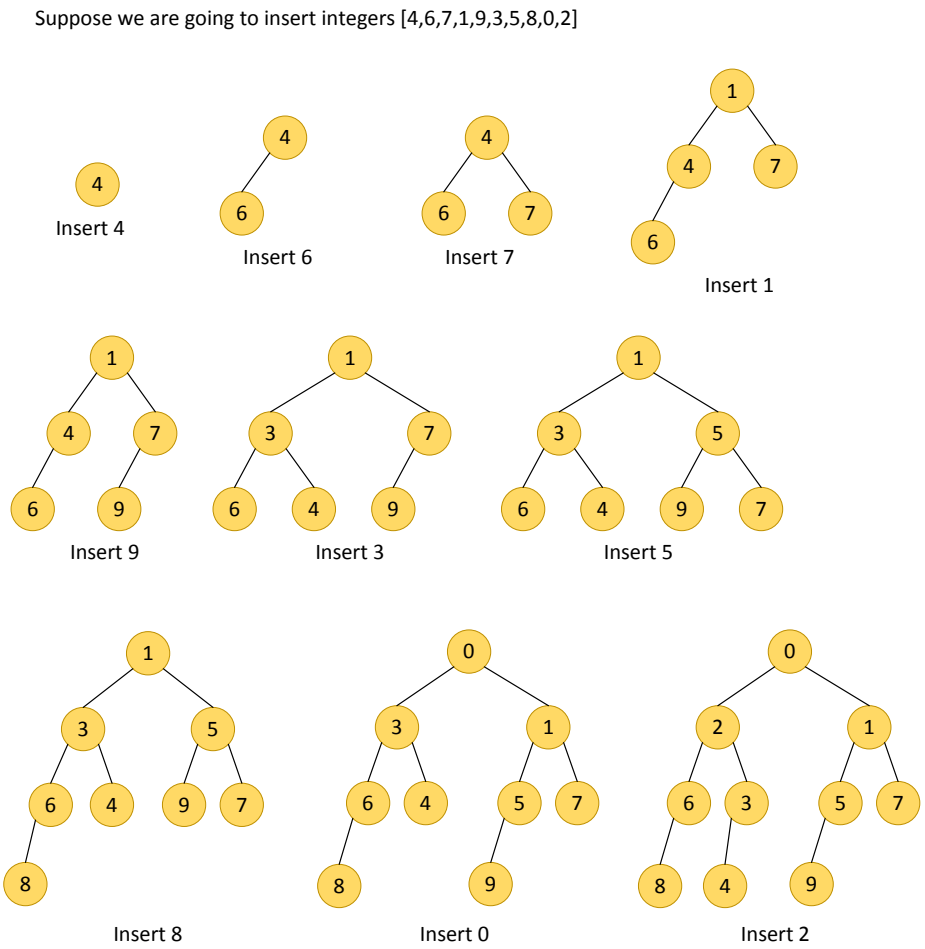

Figure 4: Steps show how to insert nodes into my min heap

Consider remove min, I will extract node's smaller child up, then continuing do this:

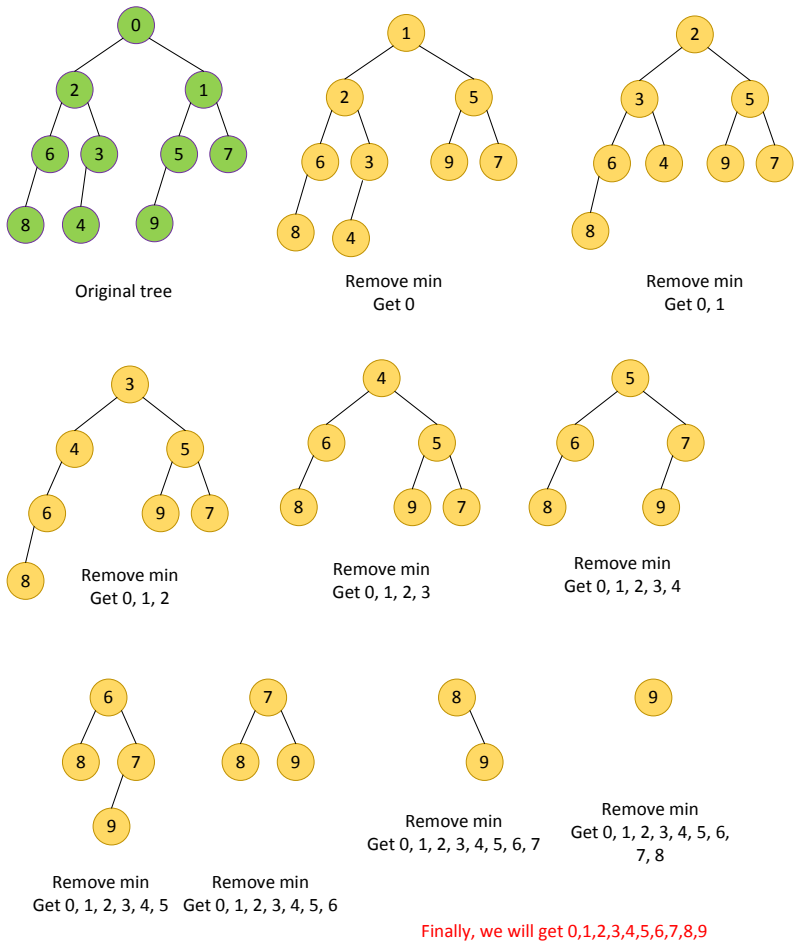

Figure 5: Steps show how to remove min from my min heap

6 Expected Performance

Though fibonacci heap's actual cost is high, its amortized cost will be much better if we execute cascading cut and pairwise combine, the resulted time complexity for prim's algorithm could be $O(nlogn + e)$. On the other hand, simple scheme can use $O(n^2)$ time complexity. When the graph is dense, they will have the same performance, but when the graph is sparse, then fibonacci heap should work much better than an array.

	actual	amortized
Insert	O(1)	O(1)
Remove min (or max)	O(n)	O(log n)
Meld	O(1)	O(1)
Remove	O(n)	O(log n)
Decrease key (or increase)	O(n)	O(1)

Fibonacci heap

- Array
  O(n^2) overall complexity
- Min heap
  O(nlogn + elogn) overall complexity
- Fibonacci heap
  O(nlogn+e) overall complexity

7 Experimental Results

I measure my program's performance only in random mode, generate 10 connected undirected graphs with different edge densities (10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%, 90%, 100%) for each of the cases n = 1000, 3000, and 5000. In order to get reliable result, I run my program 5 times for each case specified (delete smallest one and largest one) and average them by dividing 3. The follwing figures show the result:

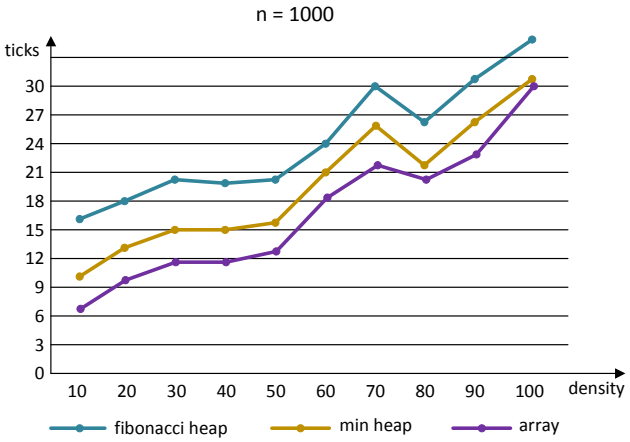

Figure 6: when n equals to 1000

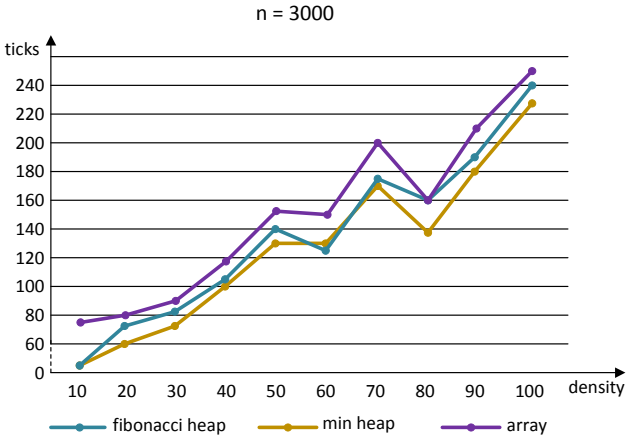

Figure 7: when n equals to 3000

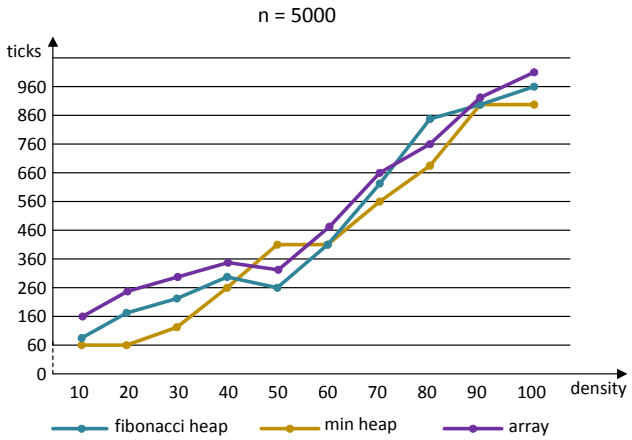

Figure 8: when n equals to 5000

As you can see, when $n = 1000$, fibonacci heap and min heap are both less efficient than array, when $n = 3000$ and $n = 5000$, fibonacci heap and min heap works a little better than simple array.

8 Analysis on Experimental Results

We have already know the corresponding time complexity for three methods, but why the result is not as we expected. After carefully examine the code, I thought the **new** operation for **TreeNode** is the most costly operation. I will later try to solve this project in C++, hope to see the result as we expected.