Advanced Data Structures Assignment #1
COP5536 Spring2014

Name: Abhishek Singh Panesar
UFID: 5111-8921
Email: asp.abhi18@ufl.edu

Index-

1. Compiling Instructions:

The Project has been compiled and tested under the following platform:

OS	Platform/OS Compiler
Mac OSX 10.8.5	Mac (Terminal)-Server / thunder.cise.ufl.edu
	Mac -Eclipse

Steps to execute the project in Server(thunder.cise.ufl.edu) using javac :

1.Open the Server

2.Compile "Dijkstra.java" using command " javac Dijkstra.java " .

3.Now run the file through command " java Dijkstra.java ".

4.For Simple Scheme input mode type the following command

java Dijkstra -s filename
    eg
    java Dijkstra -s input.txt

5.For Fibonacci Scheme input mode type the following command

java Dijkstra -f filename
    eg
    java Dijkstra -f input.txt

NOTE : IF HEAP MEMORY ERROR OCCUR PLEASE CLOSE THE SERVER AND EXECUTE AGAIN.

Steps to execute the project in Eclipse using javac :

1.Create a project
2.Open the file 'Dijkstra.java' in the project.
3.Set Run configurations arguments for input mode under run tab
    eg : -s input.txt
4.Run the program.

2. Class Description, Function and Structure overview :

Dijkstra
    Edge
        public Edge(int point, int weight)
        void setEP(int e)
        void setWeight(int weight)
        int getEndPoint()
        int getWeight()

ArrayList<A>

- **void add(A e)**
- **void nulify()**
- **int posOf(Object o)**
- **int size()**
- **void verifysize()**

SimpleDijkstra

- **void dijkstra()**
- **void initialhelpvector()**
- **updateNeighbors(ArrayList<Edge> neighbors,int node)**
- **int ClosestUnvisited()**
- **void cleanup()**

node

fibonacci

- **void nulify()**
- **boolean isEmpty()**
- **void insert(node n, int v)**
- **void decreasekey(node n, int k)**
- **void cut(node n, node p)**
- **void cascadingcut(node n)**
- **node removemin()**
- **void consolidate()**
- **link(node n, node par)**
- **int rread(int node, int d)**

FibonacciDijkstra

- **void dijkstra()**
- **void updateNeighbors(ArrayList<Edge> n,int node)**
- **void initialhelpvector()**
- **void cleanup()**
- **void check(int x,int c)**
- **int ClosestUnvisited()**
- **int readfile(String file) throws IOException**

Main

```
SimpleDijkstra.cleanup();
FibonacciDijkstra.cleanup();
FibonacciDijkstra.dijkstra();
 FibonacciDijkstra.arrayofnode.nulify();
 FibonacciDijkstra.dist.nulify();
 FibonacciDijkstra.predecessor.nulify();
 FibonacciDijkstra.visited.nulify();
 SimpleDijkstra.cleanup();
FibonacciDijkstra.dijkstra();
FibonacciDijkstra.arrayofnode.nulify();
FibonacciDijkstra.dist.nulify();
 FibonacciDijkstra.predecessor.nulify();
FibonacciDijkstra.visited.nulify();
```

3. Detailed Description

Dijkstra is the main execution class of the program which contains the functions for implementing simple dijkstra algorithm, dijkstra algorithm using simple scheme and dijkstra algorithm using Fibonacci heap. Now each class and function is described below in detail:

Classes and the functions associated with it:

class Edge : Edge class represents the adjacency list which returns the endpoint as well the weight of that edge.

public Edge(int point, int weight): It is the constructor of the edge class.
public void setEP(int e): It is the method that sets the end point
public void setWeight(int weight): It is the method that sets the weight of the edge.
public int getEndPoint(): It the method to return the endpoint of the edge.
public int getWeight(): It is the method to return the weight of the edge.

class ArrayList<A> : Class ArrayList<A> representes the properties of arraylist.

public ArrayList(): It is the constructor of the ArrayList class.

class SimpleDijkstra: class SimpleDijkstra implements the dijkstra algorithm using simple scheme.

void dijkstra(): It is the function that implements dijkstra algorithm.
void initializeHelpervectors(): It is used to initialise the value of the class members to their default value.
updateNeighbors(listofarrays<Edge> neighbors,int node): It represents the neighbouring nodes and the node under consideration as its argument and then scans through all the nodes in the adjacency list of the node under consideration.
int ClosestUnvisited() : It checks the visited list and then returns the index of the nodes are not visited currently.
void cleanup(): It cleans up all the data in the lists so as to implement the next round of the dijkstra algorithm.

Class node: class node defines the structure of the Fibonacci heap that is implemented using a doubly circular linked list which has the following fields in it: parent, child, left, right.

class Fibonacci: class Fibonacci implements the Fibonacci heap data structure and perform various operations.

public fibonacci(): It is the constructor of the Fibonacci heap.
void nulify(): It clears all teh variables.
boolean isEmpty(): It is used to check whether the Fibonacci heap is empty or not.
insert(node n, int v): It inserts a node into the Fibonacci heap and sets the key of this node.
decreasekey(node n, int k): It descreases the key of the node to the value passed in the argument.
cut(node n, node p): It removes the node from the siblings list whose key is decreased.

cascadingcut(node n): It node removemin(): It removes the minimum keyed node from the Fibonacci tree.
consolidate(): It scans the top level tree and combines the two nodes who have the same degree.
void link(node n, node par): It combines the two nodes.

class FibonacciDijkstra: class FibonacciDijkstra implements Dijkstra algorithm using Fibonacci heap.
void updateNeighbors(ArrayList<Edge> n,int node): It takes the collection of edges that represents the neighbouring nodes.
void initialhelpvector(): It initialise the value of the class members to their default values.
void cleanup(): It cleans the data present in the various lists.
ClosestUnvisited(): It returns the index of the nodes that are not visited currently.

Class Main: In class Main the functions are invoked.

4.User Input mode Results

```
thunder:34% java Dijkstra -s inp.txt

Source is 0

Total no of vertices is 8

No of edges is 28

Vertices  Distance from source
0         0
1         4
2         12
3         19
4         21
5         11
6         9
7         8
8         14
thunder:35%
```

```
thunder:39% java Dijkstra -s Input.txt

Source is 0

Total no of vertices is 3

No of edges is 3

Vertices  Distance from source
0         0
1         3
2         1
thunder:40%
```

```
thunder:38% java Dijkstra -f Input.txt

Source is 0

Total no of vertices is 3

No of edges is 3

Vertices  Distance from source
0         0
1         3
2         1
thunder:39%
```

```
thunder:35% java Dijkstra -f inp.txt

Source is 0

Total no of vertices is 8

No of edges is 28

Vertices  Distance from source
0         0
1         4
2         12
3         19
4         21
5         11
6         9
7         8
8         14
thunder:36%
```

5.Expected Output

While performing the program i expected that Fibonacci will be better for denser graphs because the complexity of Fibonacci heap is O(nlogn + e) whereas complexity of simple scheme is $O(n^2)$.

6. Comparison

N=1000

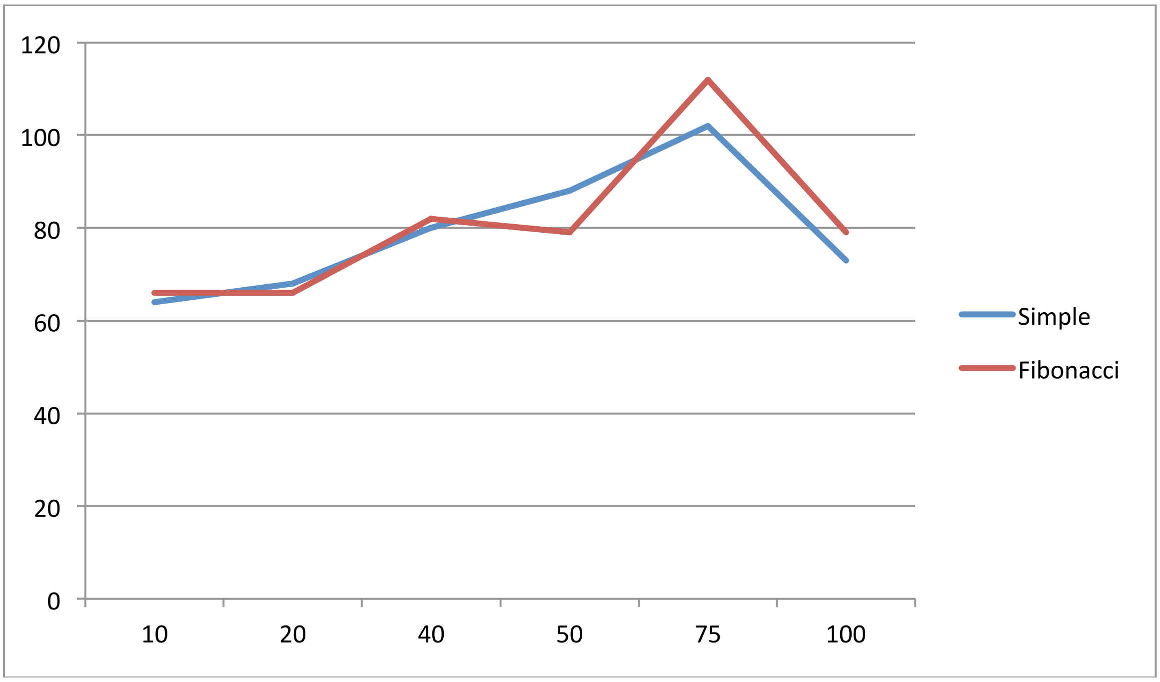

Figure 1: Performance Analysis for N=1000

```
thunder:17% java Dijkstra -r 1000 20 24
Number of vertices   Density     Simple scheme(msec)   F-heap scheme(msec)
Loading.........

1000                    20.0%         68                    66
thunder:18% java Dijkstra -r 1000 50 24
Number of vertices   Density     Simple scheme(msec)   F-heap scheme(msec)
Loading.........

1000                    50.0%         88                    79
thunder:19% java Dijkstra -r 1000 75 24
Number of vertices   Density     Simple scheme(msec)   F-heap scheme(msec)
Loading.........

1000                    75.0%         102                   112
thunder:20% java Dijkstra -r 1000 100 24
Number of vertices   Density     Simple scheme(msec)   F-heap scheme(msec)
Loading.........

1000                    100.0%        73                    79
thunder:21% java Dijkstra -r 1000 10 24
Number of vertices   Density     Simple scheme(msec)   F-heap scheme(msec)
Loading.........

1000                    10.0%         64                    66
thunder:22% java Dijkstra -r 1000 40 24
Number of vertices   Density     Simple scheme(msec)   F-heap scheme(msec)
Loading.........

1000         _          40.0%         80                    82
```

d	simple	f-heap
10	64	66
20	68	66
40	80	82
50	88	79
75	102	112
100	73	79

N=3000

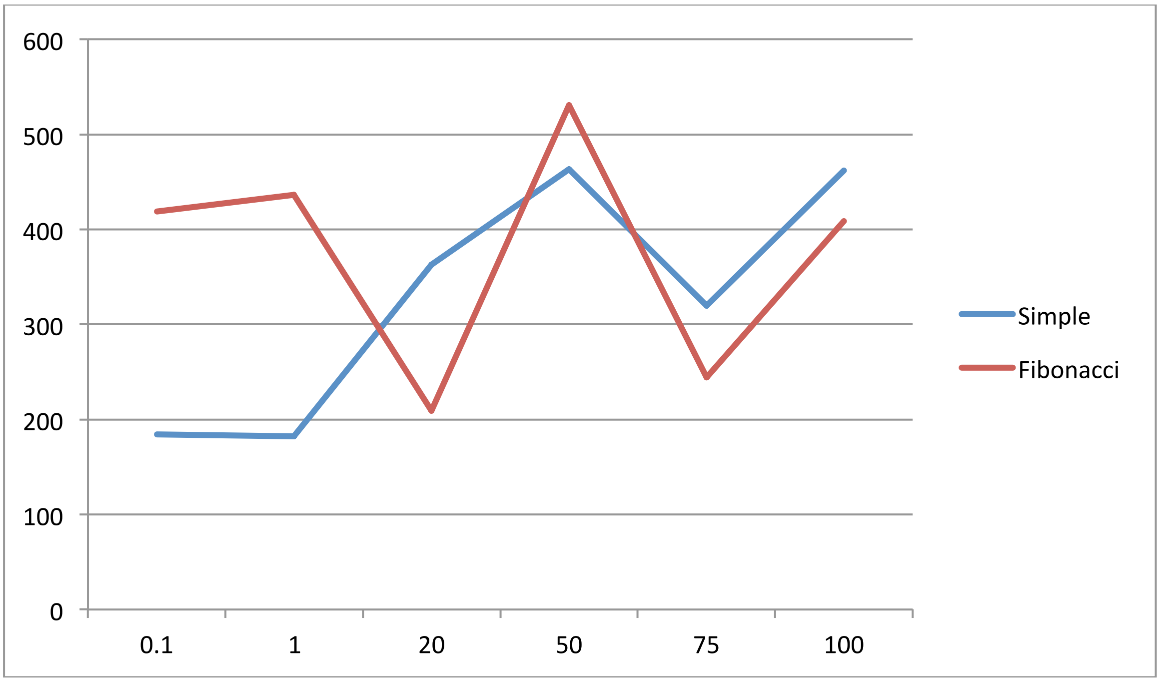

Figure 2: Performance Analysis for N=3000

```
thunder:23% java Dijkstra -r 3000 1 24
Number of vertices    Density     Simple scheme(msec)    F-heap scheme(msec)
Loading.........

3000                    1.0%            182                    436
thunder:24% java Dijkstra -r 3000 .1 24
Number of vertices    Density     Simple scheme(msec)    F-heap scheme(msec)
Loading.........

3000                    0.1%            184                    419
thunder:25% java Dijkstra -r 3000 20 24
Number of vertices    Density     Simple scheme(msec)    F-heap scheme(msec)
Loading.........

3000                    20.0%           363                    209
thunder:26% java Dijkstra -r 3000 50 24
Number of vertices    Density     Simple scheme(msec)    F-heap scheme(msec)
Loading.........

3000                    50.0%           463                    531
thunder:27% java Dijkstra -r 3000 50 24
Number of vertices    Density     Simple scheme(msec)    F-heap scheme(msec)
Loading.........

3000                    50.0%           456                    419
thunder:28% java Dijkstra -r 3000 75 24
Number of vertices    Density     Simple scheme(msec)    F-heap scheme(msec)
Loading.........

3000                    75.0%           320                    244
thunder:29% java Dijkstra -r 3000 100 24
Number of vertices    Density     Simple scheme(msec)    F-heap scheme(msec)
Loading.........

3000                    100.0%          462                    409
thunder:30% java Dijkstra -r 3000 40 24
Number of vertices    Density     Simple scheme(msec)    F-heap scheme(msec)
Loading.........

3000                    40.0%           401                    341
thunder:31%
```

d	simple	f-heap
0.1	184	419
1	182	436
20	363	209
50	463	531
75	320	244
100	462	409

N=5000

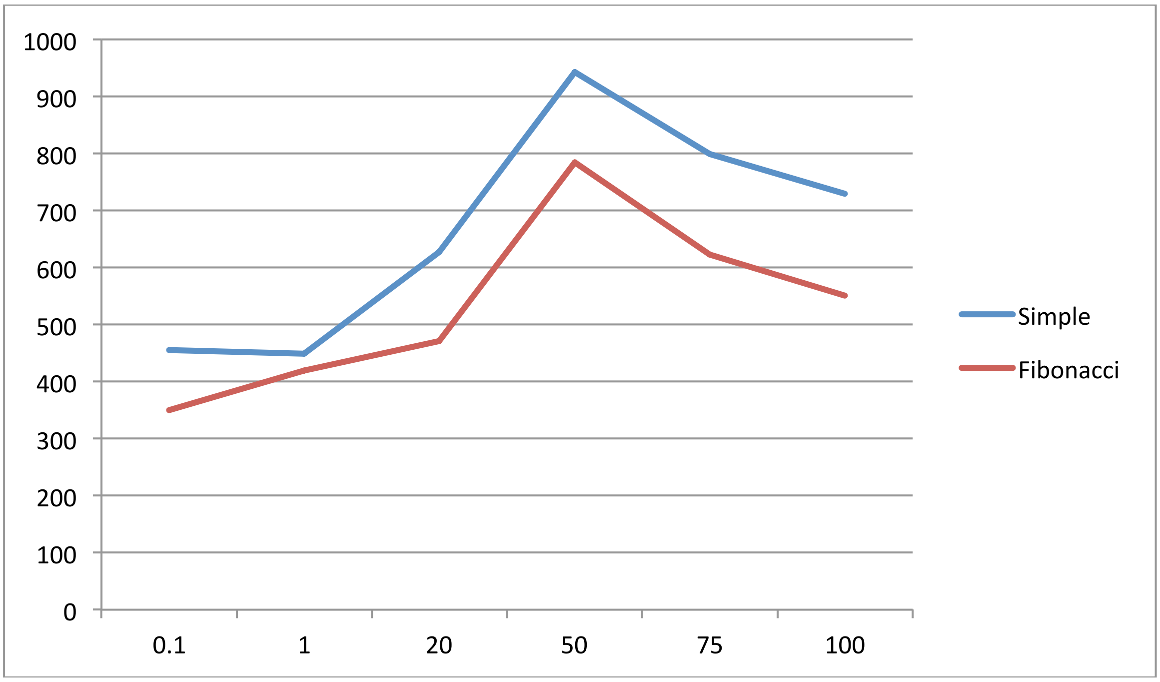

Figure 3:Performance Analysis for N=5000

```
thunder:34% java Dijkstra -r 5000 .1 24
Number of vertices   Density     Simple scheme(msec)   F-heap scheme(msec)
Loading.........

5000                 0.1%        455                   350
thunder:35% java Dijkstra -r 5000 .1 24
Number of vertices   Density     Simple scheme(msec)   F-heap scheme(msec)
Loading.........

5000                 0.1%        456                   360
thunder:36% java Dijkstra -r 5000 1 24
Number of vertices   Density     Simple scheme(msec)   F-heap scheme(msec)
Loading.........

5000                 1.0%        449                   419
thunder:37% java Dijkstra -r 5000 20 24
Number of vertices   Density     Simple scheme(msec)   F-heap scheme(msec)
Loading.........

5000                 20.0%       627                   471
thunder:38% java Dijkstra -r 5000 50 24
Number of vertices   Density     Simple scheme(msec)   F-heap scheme(msec)
Loading.........

5000                 50.0%       943                   784
thunder:39% java Dijkstra -r 5000 75 24
Number of vertices   Density     Simple scheme(msec)   F-heap scheme(msec)
Loading.........

5000                 75.0%       799                   623
thunder:40% java Dijkstra -r 5000 100 24
Number of vertices   Density     Simple scheme(msec)   F-heap scheme(msec)
Loading.........

5000                 100.0%      729                   551
thunder:41%
```

d	simple	f-heap
0.1	455	350
1	449	419
20	627	471
50	943	784
75	799	623
100	729	551

7. Conclusion Of Analysis

On testing the program for lower densities that is for density 1-20% , I found that simple scheme performs faster and better than the Fibonacci heap and as we increase the densities the performance of Fibonacci tends to increase at a very good rare. Therefore, we conclude that for denser graphs Fibonacci performs better than the simple scheme and for sparse graph simple scheme performs better than the Fibonacci heap.