Advanced Data Structures
Project Report

Name: RajaDinesh Kumar Murugesan

UFID: 41181658

Email:mrajadinesh@ufl.edu

Package Details:

The project was developed in Net Beans IDE 7.4 with JDK7.

The Zip file attached contains the following files:

- Source Code
- Executable Files
- dijikstra (NetBeans project file)
- Javadoc
- Advanced Data Structures_Report_RajaDinesh Kumar M.pdf

Opening a javadoc:

The javadoc contains the Class and the method summary of the project. It is located in the following path:

- *javadoc\index.html*

Instructions for running the project:

From Net Beans:

Open the package dijikstra from the zip from NetBeans(File->Open Project)

To run in random mode:

- Go to Run -> Set Project Configuration -> Customize
- In the Arguments section, type -r n d x (Ex: -r 1000 100 0) to run the program in the random mode. This mode performs dijikstra algorithm using both the simple heap and the Fibonacci heap and displays the execution time for both the heaps.

To run in user input mode:

- Go to Run -> Set Project Configuration -> Customize

Note: Please place the program and the input file in the same folder.

Dijikstra using simple heap:

- In the Arguments section, type –s filename (e.g. –s test.txt) to run the program for performing dijikstra algorithm using simple heap.

Dijikstra using Fibonacci Heap:

- In the Arguments section, type –f filename (e.g. –f test.txt) to run the program for performing dijikstra algorithm using fibonacci heap.

Note:
If the IDE throws insufficient heap error or GC limit exceeded error, increase the heap space for the program by performing the below steps:

- Go to Run -> Set Project Configuration -> Customize
- In the VM options, type -Xmx4096N to allocate additional heap space for the program and execute the code again.

Compiling the project from the command line in Linux environment:

Go to the location of the Source code in the folder by using the following command:

cd "Source Code"

Type the following command for compiling the project

javac dijikstra.java

And type the following commands for running the program in various mode:

1. java dijikstra –r 1000 0 100 (Random Mode)
2. java dijikstra –s filepath (File input using simple Heap)
    - ex: java dijikstra –s test.txt
3. java dijikstra –f filepath (File input using Fibonacci Heap)
    - ex: java dijikstra –f test.txt

Running the project from the command line in Linux environment:

Go to the location of the executable files in the folder by using the following command:

cd "Executable Files"

And type the following commands for running the program in various mode:

1. java dijikstra –r 1000 0 100 (Random Mode)
2. java dijikstra –s filepath (File input using simple Heap)
    - ex: java dijikstra –s test.txt
3. java dijikstra –f filepath (File input using Fibonacci Heap)
    - ex: java dijikstra –f test.txt

Performance Comparison:

Expected Performance:

The overall complexity for calculating the minimum distance in dijikstra algorithm using simple scheme is expected to be of $O(n^2)$. While, for Fibonacci heap, the overall complexity is expected to be of O(nlog n + e), as the amortized complexity of removeMin, decreaseKey are O(logN) and O(1) respectively.

- n - Number of nodes in the graph.
- e - Number of edges in the graph

The density of the graph is the main factor, which influences the performance of both the schemes. If the density of the graph increases, the number of edges in the graph increases, which in turn increases the complexity of Fibonacci heap, which is $O(nlog\ n + n^2)$. Whereas, the complexity of Simple heap remains the same, which is $O(n^2)$. Hence, when the density increases, the performance of Fibonacci heap decreases and is close to the performance of Simple heap.

Actual Performance:

The simple scheme uses findMin() method, to find the minimum distance to reach each node from the source node for a given graph using dijikstra algorithm, whose performance is of $O(n^2)$. In worst case, the method has to traverse the entire node for finding the edge with next minimum distance, which is of O(n). So the overall complexity becomes $O(n^3)$ in worst case. Whereas the worst case performance of Fibonacci Heap is O(nlog n + e) for the graph with the small edge count. When the number of edges increases, the worst case performance becomes $O(nlog\ n + n^2)$, which is close to $O(n^3)$.

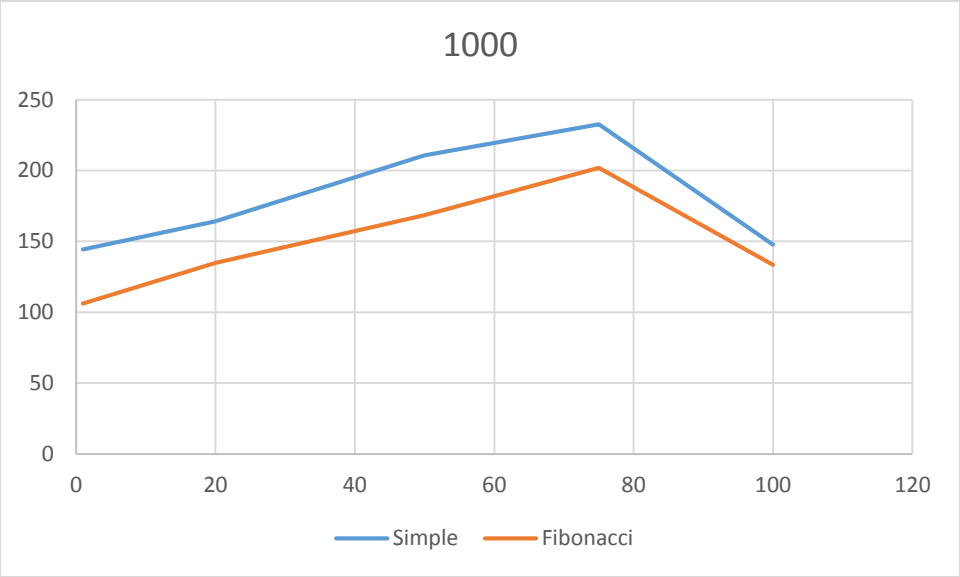
The performance analysis of both simple and Fibonacci heap, for different number of nodes and edges is shown below. The x-axis in the graph denotes the density of the graph and the y-axis denotes the average running time of both the schemes in milliseconds.

Performance Analysis:

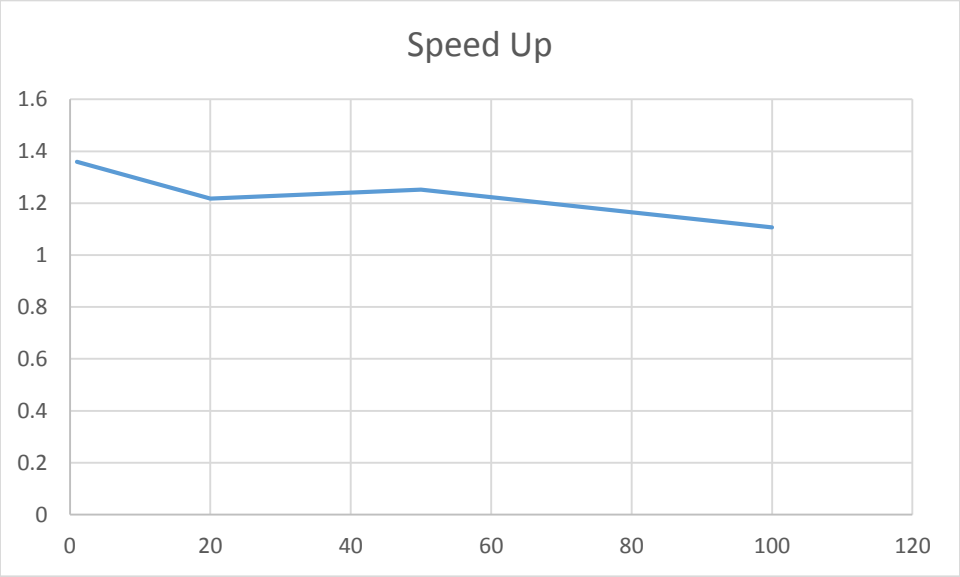
Speed up = Average Run time for Simple Heap/Average Run time for Fibonacci Heap

Graphs with 1000 Nodes

	Average Run time in milliseconds	
Density	Simple Heap	Fibonacci Heap
1	144.4	106.2
20	164.2	134.8
50	210.8	168.4
75	232.6	202
100	147.8	133.4



- X axis – Density
- Y axis – Average Runtime in milliseconds

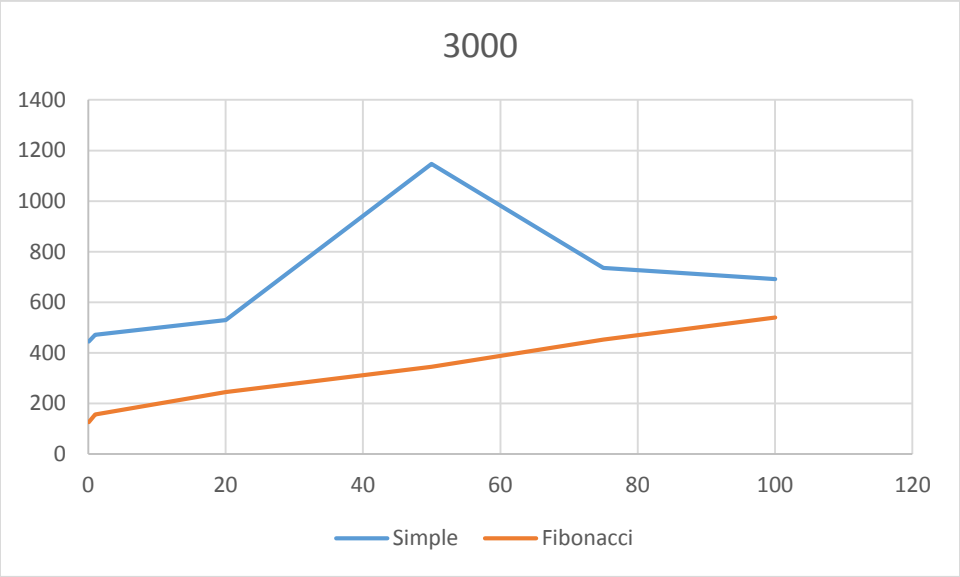


- x-axis – Density
- y-axis – Speed up

Note: The graph is generated based on 10 output readings, with the program being run on cise linux machine

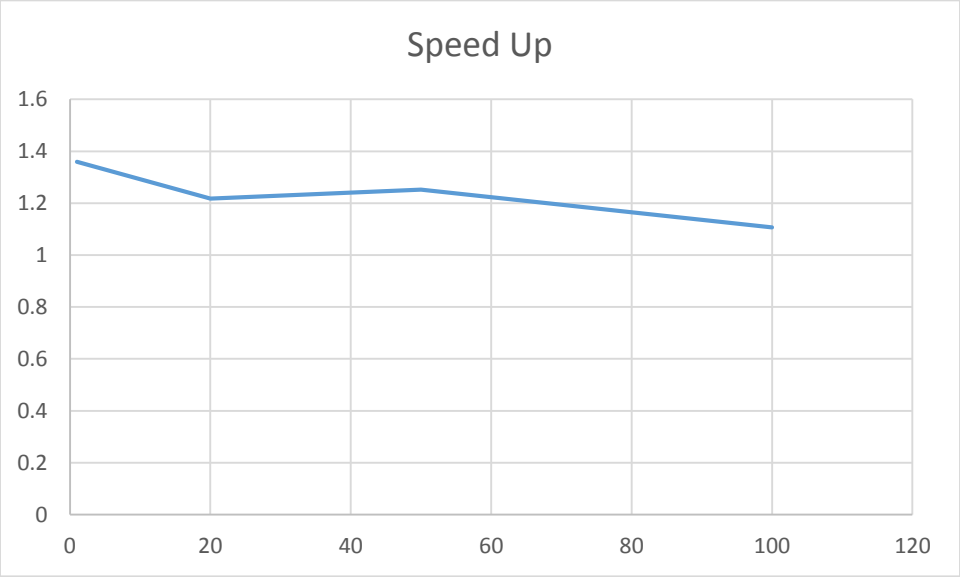
Graph with 3000 nodes:

	Average Run time in milliseconds	
Density	Simple Heap	Fibonacci Heap
0.1	445.4	126.8
1	472	156.4
20	530	245.6
50	1146.8	345
75	736.4	453
100	692.2	540.4



- X axis – Density
- Y axis – Average Runtime in milliseconds

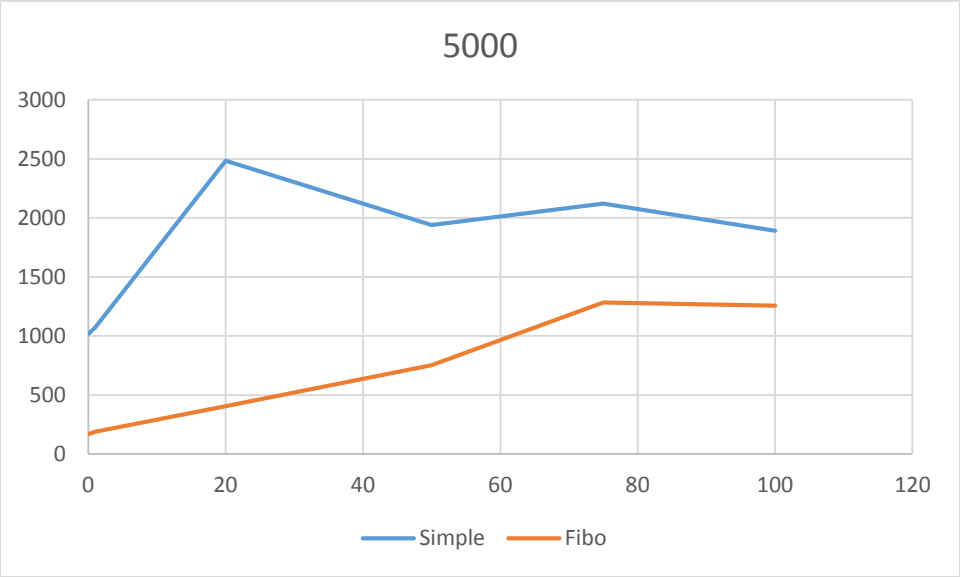
Speed Up:



- x-axis – Density
- y-axis – Speed up

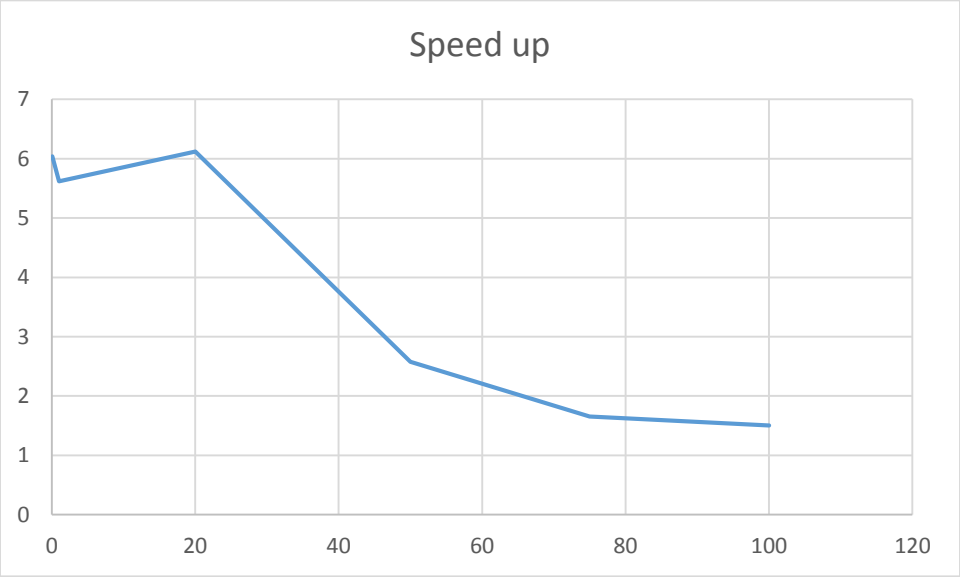
Graph with 5000 nodes:

	Average Run time in milliseconds	
Density	Simple Heap	Fibonacci Heap
0.1	1021.4	169
1	1072.2	190.6
20	2484.8	405.8
50	1940	753.6
75	2121.6	1283
100	1892	1257.2



- X axis – Density
- Y axis – Average Runtime in milliseconds

Speed Up:



- x-axis – Density
- y-axis – Speed up

Conclusion:

The performance of Fibonacci heap decreases, when the density of the graph increases and is close to the performance of simple heap, which advocates the claim made in performance comparison section.

Function prototypes:

The function prototypes are explained separately in a javadoc, which is located in the following path:

- *javadoc\index.html*

Program Structure:

1. The graph can be generated randomly or it can be constructed from the file provided by the user.
2. The graph is generated based on the user input.
3. Call the function createGraph for generating graph from the random numbers.
4. The function createGraph generates the edges and the cost randomly and adds them to the graph, only if the graph is empty or if the generated edges are not already in the graph.
5. After the graph is generated, the dfs function is called to check whether all the edges are connected in the generated graph.
6. Once all the edges are connected in the graph, the function dijikstraSimpleHeap is called, for performing dijkstra algorithm using simple heap and the function dijkstraFibonacci is called, for performing dijkstra algorithm using Fibonacci heap.
7. Once the dijikstra algorithm is completed using both the schemes, the running time for both the schemes are displayed in milliseconds.
8. In the user input mode, the graph is generated from the input file, by calling the method createGraphfromFile(FileName).
9. After the graph is generated, the dfs function is called to check whether all the edges are connected in the generated graph.
10. Once all the edges are connected in the graph, the function dijikstraSimpleHeap or dijkstraFibonacci is called, based on the user input.
11. Once the dijikstra algorithm is completed, the minimum distance to each node from the source and the running time for the algorithm will be displayed.