# Advaced Data Structrue (COP 5526)

# Project Report

Name: Jiaqi Zhu

UFID: 79199179

Email: zhujiaqi@ufl.edu

# 1. Compile and Run

After unzip the Jiaqi_Zhu.zip, the folder will have 4 Java source file: Dijkstra.java, Algorithm.java, Utility.java and FHeap.java.

To compile the files in Unix environment, first move to the right directory, then use:

*javac *.java*

After compiled, use following command to run the program:

*java Dijkstra –r (# of nodes) (density percent) (source node)*

Generate a random graph with given number of nodes and density, notice that actually the source node is useless, just make sure it will not exceed the number of nodes.

*java Dijkstra –s (filename.txt)*

Read input from file and utilize simple scheme, make sure the input have the same format as the project instruction.

*java Dijkstra –f (filename.txt)*

Read input from file and utilize f-heap scheme

The folder will also contain a file called Ads.jar, which is already compiled and packed. To run the program from this file, use command:

*java –cp .:Ads.jar Dijkstra –r n d x*

*java –cp .:Ads.jar Dijkstra –s filename.txt*

*java –cp .:Ads.jar Dijkstra –f filename.txt*

# 2. Structure of the Program

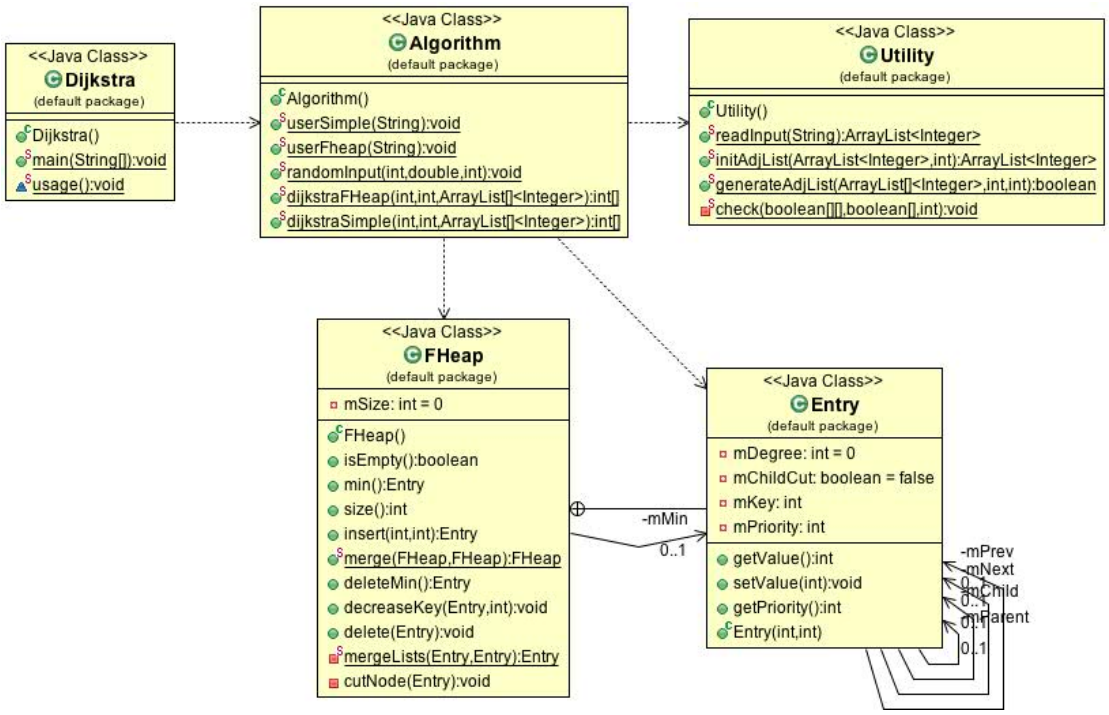


Figure 1 UML

The main method is in the Dijkstra class, in this class it will determined whether we are in User Input Mode or Random Mode and then call those function accordingly. If in User Input Mode, it will call userSimple or userFheap method in Algorithm class, otherwise it will call randomInput method.

**Class Utility**

readInput: read input from a text file and store it in a ArrayList

initAdjList: after read input from file, transfer the list of input number to Adjacency List, need to know number of nodes as input, notice that Adjacency List is implemented as an array of ArrayList

generateAdjList: for Random Mode, generate random graph and represent it as Adjacency List, check whether the random graph is connected and return the Boolean value.

check: check whether a given graph is connected, return an array of whether each node is reachable from the given source node. This method is called by generateAdjList.

**Class Algorithm**

userSimple & userFheap: call readInput and initAdjList in Utility class, and then call dijkstraSimple and dijkstraFHeap in Algorithm class accordingly.

randomInput: call generateAdjList in Utility Class, and then call both dijkstraSimple and

dijkstraFHeap to compare their running time

dijkstraSimple: given source node, number of nodes and the Adjacency List, utilize simple scheme to solve the problem

dijkstraFHeap: given source node, number of nodes and the Adjacency List, utilize f-heap scheme to solve the problem

**Class FHeap & Entry**

Entry is an inner class of FHeap and it specify the entry of Fibonacci Heap

FHeap Class is the class for Fibonacci Heap, which implement the insert, delete, deleteMin, decreaseKey and merge operations of Fibonacci Heap with some other helper functions.

# 3. Analysis of Complexity

The Complexity of the Simple scheme to find the shortest path using Dijkstra's algorithm is $O(N^2)$ and the complexity of F-Heap scheme is $O(N\log(N)+E)$ where $E$ is the number of edges in the graph and $N$ is the number of nodes.

We can expect that the performance of F-Heap scheme is comparable to Simple scheme under different value of density. And, when node become more and more, F-Heap will show benefit if density is low.

# 4. Result of Experiment

Table 1 Test Enviroment

| | |
|---|---|
| CPU | Core 2 Duo, 2GHz |
| Memory | 4Gb |
| OS | Mac OS 10.9 |

Table 2 1000 Nodes

| Density (%) | Generation Time | Simple Scheme | F-Heap Scheme |
|---|---|---|---|
| 1 | 61 | 35 | 87 |
| 20 | 174 | 65 | 123 |
| 50 | 317 | 55 | 116 |
| 75 | 499 | 58 | 112 |

| | | | |
|---|---|---|---|
| 100 | 271 | 66 | 124 |

Table 3 3000 Nodes

| Density (%) | Generation Time | Simple Scheme | F-Heap Scheme |
|---|---|---|---|
| 0.1 | 106 | 86 | 102 |
| 1 | 170 | 106 | 107 |
| 20 | 2572 | 131 | 172 |
| 50 | 5462 | 171 | 272 |
| 75 | 10470 | 190 | 339 |
| 100 | 8511 | 196 | 346 |

Table 4 5000 Nodes

| Density (%) | Generation Time | Simple Scheme | F-Heap Scheme |
|---|---|---|---|
| 0.1 | 214 | 208 | 133 |
| 1 | 373 | 198 | 147 |
| 20 | 5883 | 269 | 306 |
| 50 | 20438 | 428 | 675 |
| 75 | 39074 | 616 | 1009 |

Notice that 1000 nodes with 0.1% density cannot generate connected graph. 5000 nodes with 100% density often gives out of memory exceptions and thus hard to calculate average running time.

Generation time is often very large. When density is 100%, we know all the edge should be in the graph, so I just randomly give each edge a cost but not randomly pick edges, which makes the complexity of generation be $O(N^2)$. However it is still running very slow, maybe the java.util.ArrayList is not so efficiency and it maybe also the reason that F-heap scheme is not so fast.

5000 nodes with 0.1% and 1% density problem will be solved by F-Heap scheme faster. So I test some low density problem additionally.

Table 5 Density 1%

| # of Nodes | 1000 | 2000 | 3000 | 4000 | 5000 | 6000 | 7000 | 8000 | 9000 | 10000 |
|---|---|---|---|---|---|---|---|---|---|---|
| Generation | 56 | 103 | 145 | 259 | 385 | 450 | 848 | 882 | 974 | 1224 |
| Simple | 30 | 55 | 104 | 138 | 198 | 285 | 365 | 461 | 596 | 705 |
| F-Heap | 75 | 88 | 106 | 120 | 134 | 163 | 161 | 167 | 216 | 217 |

It seem that given a certain density, the running time of F-Heap scheme increase slowly than the Simple scheme as the number of nodes increase.