NAME: MANASI SHARMA

UFID: 90009559

EMAIL: manasi@cise.ufl.edu

Compiler used:

JAVA SE 1.8

Command to compile:

javac dijkstra.java

Commands to run:

For random mode: java dijkstra –r n d x where 'n', 'd' and 'x' are integer values

For user input mode simple scheme: java dijkstra –s input1.txt

For user input mode fibonacci scheme: java dijkstra –f input1.txt where input1 is a string like "readme.txt" (note: .extension of file is required)

Structure of program:

Main(): Main method takes the input from the command line. Stores in the corresponding variables. Calculates the time taken by simple and Fibonacci algorithms in case of random mode.

Neighbors class: Neighbors class' objects are used as data values in a node's adjacency list. It has value and cost field for storing the second node's id and the corresponding weight.

generateGraph(): generateGraph function is responsible for generating the random graph that is generated on basis of 'n' and 'd' values supplied. The generated graph is stored in form of adjacency list i.e. a list of list where each node stores each of its neighbour in its corresponding list. Edges are generated until the density condition is reached. In some cases a complete graph may not be generated under the given density limit, please rerun the program in that case.

readFile(): readFile function reads the nodes and edge weights from a file in case of user input mode. Supplied file should contain the values as prescribed in the given description or weird graphs will be generated. The same neighbour class and adjacency list concept is used for storing the graph. readFile and generateGraph are mutually exclusive and will generate the graph in the same list variable according to the desired mode.

Random(): random() function randomly generates the graph and computes the time taken by simple and Fibonacci algorithms to generate the shortest path from a source vertex.

Two scheme in the program are

- simpleScheme
- fheapScheme

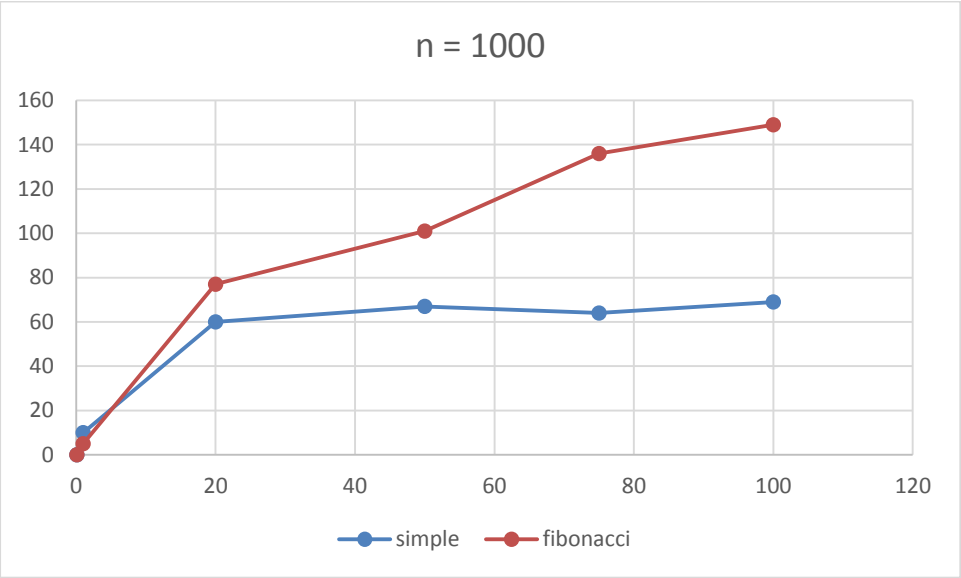
FibonacciHeap class: This class helps in managing the nodes in the Fibonacci heap. It stores the right_sibling, id, left_sibling, child, parent, child_cut, degree and priority of a node, where priority is the key of the node. Following methods are utilized for managing the Fibonacci heap.

- Insert() for inserting nodes in the heap with id and key values. Elements are inserted in circular list that stores all the roots in the heap.
- mergeList() is used for inserting a new node in an already existing circular list by changing the left and right pointers.
- cutCascade() (setting the Childcut field to true and false) is used for cutting a node from a tree and reinserting it into the root list. Accordingly changes are made in the left sibling, right sibling, parent and children of that node.
- removeMin() : theNode points to the Fibonacci heap node that contains the element that is to be removed. Thereafter, removes the element pointed by the MIN object i.e. the node with the minimum key. Left and right sibling nodes' pointers are rearranged to maintain the circular list after the removal. If the removed node had any children, they are added in the root list. After that, roots with same degree are combined together and new min node is calculated.
- decreaseKey() (inserted int top-level list) function changes the key value of a specific node to a new value supplied and if applicable, cutCascading is performed afterwards.

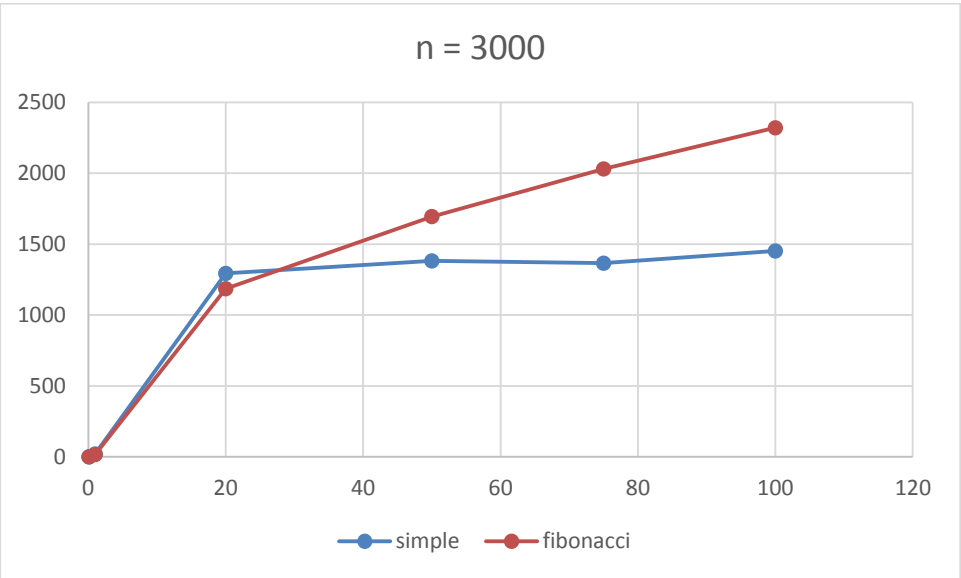
Performance for random mode operation:

Nodes	density	simple	Fibonacci
1000	0.1	0	0
	1	10	5

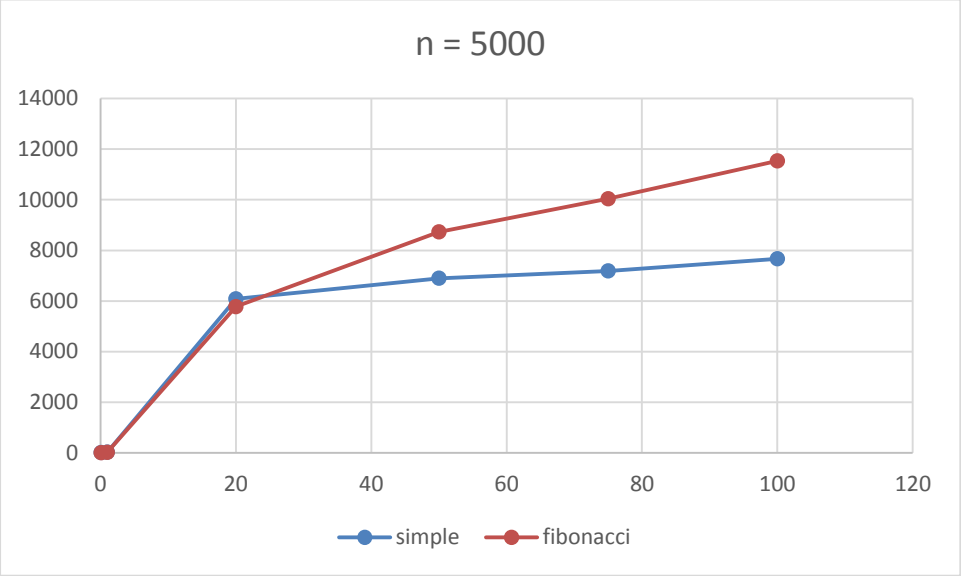
20	60	77
50	67	101
75	64	136
100	69	149



nodes	density	simple	fibonacci
3000	0.1	0	0
	1	20	16
	20	1294	1186
	50	1382	1695
	75	1366	2031
	100	1452	2321



nodes	density	simple	fibonacci
5000	0.1	13	6
	1	29	20
	20	6084	5775
	50	6899	8730
	75	7188	10038
	100	7668	11534



Expected results:

Simple scheme should generate the shortest path in $O(n^2)$ time while the Fibonacci heap should generate the path in O(nlogn+e) time.

Thus for small 'n' values, simple scheme should give better results. And as n increases, Fibonacci scheme will start performing better. It should further conclude that Fibonacci scheme works efficiently only for sparse graph and for dense graphs , simplescheme takes the lead.

Actual results and Observations:

The following observations are made:

1. As we go increasing n, Fibonacci heap performs better with large n values as n^2 greatly increases but there is not much effect on nlogn+e since e is almost negligible

2. As we go on increasing density, the number of edges increases and hence e starts playing important part in the nlogn+e formula and for enough dense graphs, simple scheme starts performing better than Fibonacci due to the n^2 logn in f heap.

Clearly, actual results nearly meet the expected ones.