Name: Risheng Wang
UFID: 82892242
UF Email Account: r.wang@ufl.edu

Note: Plots and Comments are in the last of this report.

- Compiler:
  In this project I use Eclipse to compile and run my code.
  I also test the code with Ubuntu Linux and on VM. When compiling, put all java files under the same directory and type "javac *.java" on the terminal.
  Type "java dijikstra –r n density x" to run code in random mode.
  Type "java dijikstra –s filename" to run in user mode with simple array.
  Type "java dijikstra –f filename" to run in user mode with f-heap.

- Function prototype:
  The code includes three top level classes and one nested class:
  1. dijikstra.class:
     void initiate(Integer n, double density, int source)
     // Initiation in random mode
     void initiate(String filename, String mode)
     // Initiation in user mode
     void run_Dijikstra(Vertex[] Unset)
     // Run dijikstra algorithm with simple array
     Vertex find_Min(Vertex[] Unset, int count)
     //Find minimum distance in the array and remove it out
     void relax_Path(Vertex U, Vertex V) // Relax the path to U's neighbor V
     void frun_Dijikstra(FibonacciHeap<Vertex> funset)
     // Run algorithm with f-heap
     void frelax_Path(Vertex U, Vertex V)
     // Relax the path to U's neighbor V in f-heap mode
     void reset(int source) // Reset the graph

  2. Vertex.class:
     Vertex(String name, int id, int s_id) // Constructor method of this class
     int getId() // Get the id of this node
     String toString() // Get the string presentation of this node
     int getDis() // Get distance from source to this node
     HashMap<Vertex, Integer> getNb()
     // Return the list of neighbors of this node

void setNb(Vertex node, int weight) // Set new neighbor
void setDis(int n_dis) // Set new distance
boolean isNb(Vertex node) // Check if edge exists
static boolean isConnected(Vertex source, int num)
        // Check if graph is connected

3. FibonacciHeap.class:
   FibonacciNode<T> insert(T elem, int key)
           // Insert a new node to f-heap with its key; return the node
   FibonacciNode<T> getMin() // Get min element in the f-heap
   boolean isEmpty() // Check whether heap is empty
   FibonacciNode<T> deleteMin()
           // Delete current min element from heap and return the element
   void decreaseKey(FibonacciNode<T> node, int new_key)
           // Decrease the key of the node
   void cutNode(FibonacciNode<T> node)
           // Cut the node from its parent if necessary
   static FibonacciNode<T> mergeList(FibonacciNode<T> one,
           FibonacciNode<T> two)
           // Merge two FibonacciNode together

4. FibonacciNode.class (nested class in FibonacciHeap class):
   FibonacciNode(Telem, int key) // Constructor method of this class
   T getElem() // Get the element of this node
   void setElem() // Set the element of this node
   int getKey() // Get the key of this node

- Summary:
  The complexity of Dijikstra algorithm mainly consists of two parts: *Choosing Shortest Path* and *Neighbor Path Relaxing*.
  Using simple array, it merely scans the whole unsettled node list to choose the shortest path. Therefore with n nodes, the complexity should be $O(n^2)$, or more precisely $\theta(\frac{n^2}{2})$ (because after every path choosing the size of unsettled list will decrease by one).
  While using f-heap, since it supports *deleteMin()* operation with amortized complexity $O(\log_2 n)$, checking n nodes will take $O(n \log_2 n)$ time.
  Obviously, f-heap is faster than simple array in choosing shortest path.

  In Neighbor Path Relaxing things are different. For simple array, relaxing is simply changing one field of the vertex. It does not affect node list so a single relaxing only takes O(1) time. However, *decreaseKey()* method of f-heap may trigger cascading node cut in certain time, in which case it will take $O(\log_2 n)$ time to finish a single path relaxing. Therefore, the more edges the graph consists of, the slower f-heap method will be.

In total, with small number of vertexes the simple array may be faster than f-heap. When edge density increases, the f-heap may be slower.

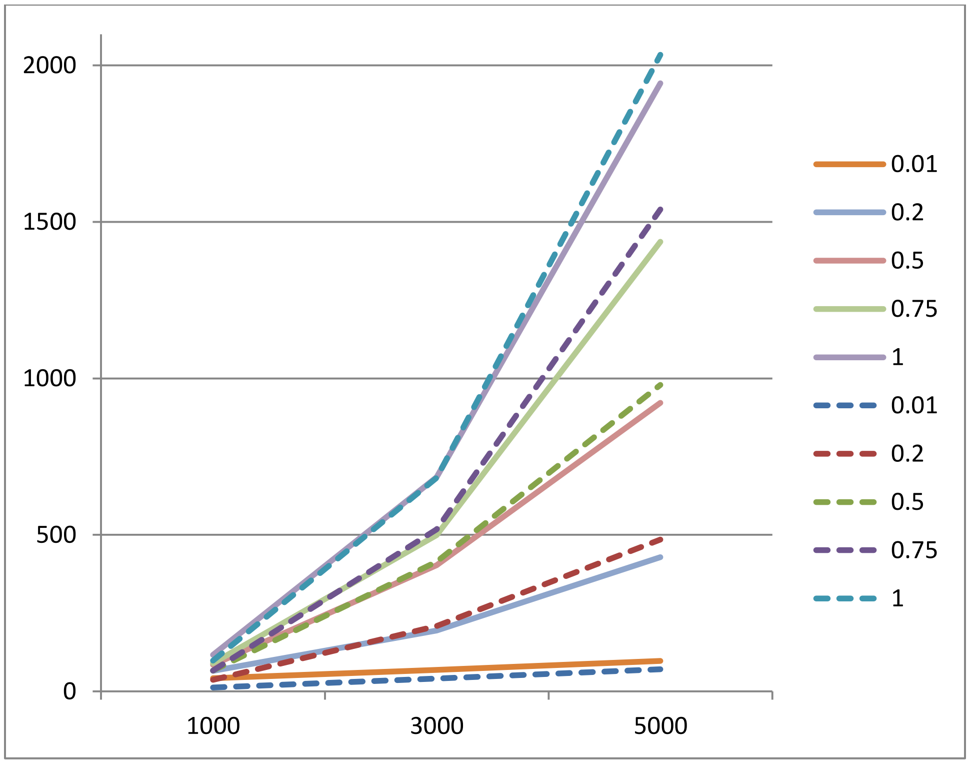
- Code Structure:

  As mentioned above, the code consists of three top level classes and one nested class.
  Vertex class represents the graph, storing node and edge information. This class keeps a static global list called *Nodes* to store the entire graph data.
  Vertex class has a field called distance to store the current distance from source to this node. This field is initiated infinity (Integer.MAX_VALUE in java) for every vertex except the source node (distance from source to source is 0).
  Every instance of Vertex keeps a neighbor map called *neighbors*. It is a HashMap, mapping neighbor vertex to the corresponding edge weight.
  Other fields: *id* is the identification of the vertex, and *name* is the string representation of the node.

  FibonacciHeap class is nested with FibonacciNode class.
  FibonacciNode is the basic unit stored in the FibonacciHeap. Each node has a parent point, a child point and two sibling points. The priority of a node is determined by its *key* field and data is stored in the *elem* field.

  dijikstra class is the main class implementing dijikstra algorithm.
  It contains two unsettled node lists using simple array and f-heap respectively, one settled node list using Arraylist and one HashMap mapping vertex to its corresponding FibonacciNode stored in the f-heap.
  If first input argument is “-r”, the algorithm will run in random mode, test simple array and f-heap, and give out the time two methods spend respectively.
  If first input argument is “-s” or “-f”, program will run in user mode, read in user supplied file and operate the algorithm using requested data structure.
  In user mode, standard output will be printed on the screen and into a file called “result_file” .

- Plots and Comments:



Comparison between s-array and f-heap

In the above picture, the dotted lines stand for s-array method, and solid lines represent f-heap method.
Although subtle difference between the results derived from two methods, it could be found that when the number of vertexes and edges is small, simple array is a bit faster than f-heap, and when this number grows up f-heap becomes superior to simple array.

This result is close to my expectation. When number of nodes is small, the difference between $\theta(\frac{n^2}{2})$ complexity and $O(n\log_2 n)$ complexity is not obvious. The dominant factor here should be disk access time. Comparing naïve scan method with *deleteMin()* algorithm, it is clear that *deleteMin()* will do more disk access to change the field of f-heap and its nodes. Every time a min element is deleted from f-heap, it will access and write new values into the fields of sibling nodes and children nodes of min element, merge top level nodes and create new child and sibling pointers. While naïve scan only reads the keys, compares the keys, and moves the smaller key to the top. Same things happen when relaxing the

path: naïve scan only changes the *distance* field of vertexes, but f-heap *decreaseKey()* will do much more. However, with the increase of number of vertexes in the graph the complexity of algorithm becomes dominant, and disk access time is less important. That is why s-array is slower than f-heap with 3000 and 5000 nodes at most time.
In fact, when I test 10000 nodes on each method, f-heap is eight times faster than s-array.

Once I thought that the larger the density of the graph, the slower f-heap method will be because there should be more cascading cuts happen when calling *decreaseKey()* method. However, it seems not the case.
In fact, consider an extreme case that the density is 100%. Now every node is connected to the remaining nodes in the graph. When calling *decreaseKey()* first time following the first *deleteMin()* operation, it will decrease all nodes' keys stored in f-heap and all nodes becomes top level skeleton trees again (because keys are initiated as infinity except the source node). Even though cascading cut happens, it only means that it does not have to cut this node again because everyone will be cut at least once. Therefore in average, *decreaseKey()* only takes O(1) time.
From the second *decreaseKey()* operation on, O(lgn) cascading cut may happen still. But actually the *decreaseKey()* operation itself will seldom be called, since in a randomly configured graph, it is higher possibility that the shorter path to the source is the closer path to the source. In the case of 100% density, the shortest path to the source may just be the edge between source and its neighbors. The higher the density is, the faster the shortest path will be found, thus reducing the chance of calling *decreaseKey()*. When *decreaseKey()* is less operated, the speed of shortest path choosing becomes dominant, where f-heap performs better than simple array.

In addition, the speed of computer and the precise of timer also affect the result. With a fast computer and millisecond level timer, it cannot tell exactly how much time the algorithm spends, especially when the graph is small and sparse. I test the code with Eclipse, and most time it throws 0 out when doing one shortest path choosing.