# <u>COP 5536 Spring 2015: Programming Project</u>

Name: Sumeet Pande

UFID: 4890-9873

## <u>*Part 1*</u>

To implement Dijkstra's Single Source Shortest Path (ssp) algorithm for undirected graphs using Fibonacci heaps. To make use the adjacency list representation for graphs.

<u>Structure & Methodology:</u>

The programming for the above part has been implemented in JAVA programming language. Hence the compiler used is the JAVA compiler for the entire code. The main idea behind this implementation is to parse the input text file which happens to store the graphical data and create a graph structure in our project consisting of nodes i.e. vertices and edges and weights for the edges. The graphical representation used is the adjacency list representation. Based on this graphical structure we have applied the Dijkstra's SSP to calculate the shortest path between any nodes/vertices. The underlying data structure used for the Dijkstra's SSP algorithm is the Fibonacci heap.

<u>Step 1:</u> The first primary requirement was to create a graph structure and to populate this graph structure by parsing the sample input files.

For this purpose we have created three different classes which are as follows:

<u>GraphVertex:</u> This class describes the attributes as well as some methods for the vertex of the graph which will be parsed from the input data.
<u>GraphEdge:</u> This class describes the attributes of the graph edge of the graph which will be parsed from the input text file.
<u>GraphStructure:</u> This graph in general describes the structure of the graph consisting of nodes and edges.

The main methods used for this are as follows:

static String[] parsing(String name)

This method basically parses the input text file line by line and populates the information of the same into a graphical structure as defined from the above three classes. This actually returns the array of strings which consist of information regarding no of nodes, no of edges and weight for each of these edges. This graph structure is then passed on to the methods responsible for computing the shortest path using Dijkstra's algorithm.

Step 2: Fibonacci heap implementation of Dijkstra’s SSP.

The underlying data structure for our implementation is the Fibonacci heap structure. For this purpose we have created two classes which are actually implementation of a node, attribute and methods of that node for the Fibonacci heap as well another class which consist of methods for different operations on Fibonacci heap. The classes are as follows:

FHNode: This class basically describes the Fibonacci heap. It describes the various attributes, constructors and node level operation methods for the Fibonacci heap data structure. The various methods used in this class are private void sibling() ,public int getData(),public void setData(int data),public int getKey(),public void setKey(int key),public FHNode getParent(),public void setParent(FHNode parent) and many other methods which perform node level operations for the Fibonacci heap structure.

FibonacciHeapDijkstra : This class was used to implement methods for operation of the Fibonacci heap. These operations are used while executing the shortes path algorithm.
The methods are as follows:

```
// method to check if the  Fibonacci heap is empty.
 public boolean isEmpty()

//Perform operations on the min element
 public FHNode findMin()
 public FHNode deleteMin()

//Insert a new vertex or node in the F heap.
   public boolean insertVertexT(FHNode node)

//Performdecrease key operationson the F-Heap
    public void decreaseKey(int dkdata, FHNode vertex)
```

Step 3: Computing the shortest path

This is accomplished by making use of multiple methods in our main class i.e. ssp.java. This main purpose here in this class is to implement methodologies to accomplish the entire program working right from inputting text file and capturing the argument which will be actually graph nodes between which the shortest path has to be calculated , the minimum weight/distance between them has to be output and the path has to be displayed.

The methodology for taking arguments is as follows:

```
//Declaring the arguements for the input text files and vertex position.
        String s= args[0];
        int source = Integer.parseInt(args[1]);
        int destination = Integer.parseInt(args[2]);
```

Once we get the input text file and the argument vertices then we implement the program code for parsing this text file and setting up Fibonacci heap data structure as mentioned in step 1 and step 2.

Once we have that up and ready we need to compute the shortest path which as follows:

```
//Computing the Djikstra's SSP
   computeShortestPath(vertices[source]);
```

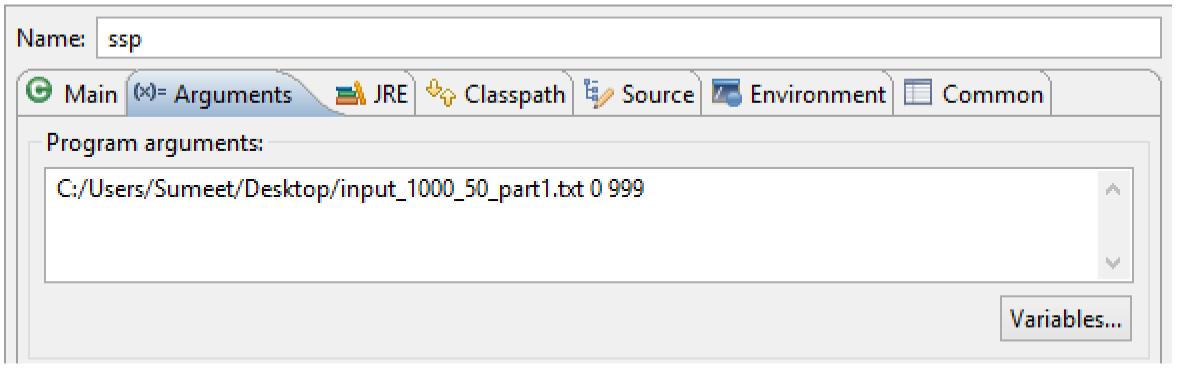
```
//Method to implement the Djikstra SSP
   public static void computeShortestPath(GraphVertex source)
```

The first method here passes on the source argument node to the method which compute shortest path. This method computes the shortest path from the source vertex to all the remaining vertices in our graph and computes the shortest path for each of them as well as update the minimum distance for each of these vertices which is initially by default set to positive infinity.

```
GraphVertex v = vertices[destination];
System.out.println(v.minDistance);
List<GraphVertex> path = getShortestPathTo(v);
System.out.println(path);
```

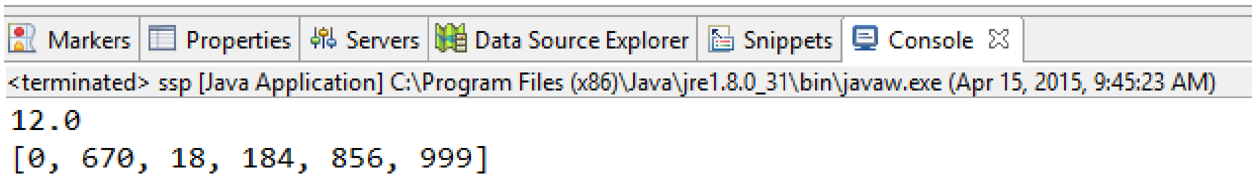
These statement then computes & displays the particular shortest path as well as the minimum distance corresponding between the source and the destination.

The parameters and the output for the same when passed the input_1000_50_part1.txt is as follows.


Name: ssp

Program arguments:

```
C:/Users/Sumeet/Desktop/input_1000_50_part1.txt 0 999
```



```
<terminated> ssp [Java Application] C:\Program Files (x86)\Java\jre1.8.0_31\bin\javaw.exe (Apr 15, 2015, 9:45:23 AM)
12.0
[0, 670, 18, 184, 856, 999]
```

## *Part 2*

Implement a routing scheme for the network using Part 1 to obtain shortest path from R to each destination router Y.

Step 1: This part is majorly similar to that of implemented in Part 1.The first primary requirement was to create a graph structure and to populate this graph structure by parsing the sample input files. Additionally we also need to parse through the files containing different IP address and converting the same to a binary string format.

For this purpose we have created three different classes which are as follows:

GraphVertex: This class describes the attributes as well as some methods for the vertex of the graph which will be parsed from the input data.

GraphEdge: This class describes the attributes of the graph edge of the graph which will be parsed from the input text file.

GraphStructure: This graph in general describes the structure of the graph consisting of nodes and edges.

In order to parse & convert the IP addresses and their binary string equivalent we have made use of Hashtable to store both the original IP and corresponding binary equivalent. The following is the declaration of the Hashtable and the method which accomplishes the task.

```
Hashtable<Integer, Double> ipTable = new Hashtable<Integer, Double>();
public Hashtable<Integer, Double> parseIPAddress(String ipname)
```

Step 2: Fibonacci heap implementation of Dijkstra's SSP.

The underlying data structure for our implementation is the Fibonacci heap structure. For this purpose we have created two classes which are actually implementation of a node, attribute and methods of that node for the Fibonacci heap as well another class which consist of methods for different operations on Fibonacci heap. The classes are as follows:

FHNode: This class basically describes the Fibonacci heap. It describes the various attributes, constructors and node level operation methods for the Fibonacci heap data structure. The various methods used in this class are private void sibling() ,public int getData(),public void setData(int data),public int getKey(),public void setKey(int key),public FHNode getParent(),public void setParent(FHNode parent) and many other methods which perform node level operations for the Fibonacci heap structure.

FibonacciHeapDijkstra : This class was used to implement methods for operation of the Fibonacci heap. These operations are used while executing the shortes path algorithm.
The methods are as follows:

```
// method to check if the  Fibonacci heap is empty.
 public boolean isEmpty()

//Perform operations on the min element
 public FHNode findMin()
 public FHNode deleteMin()

//Insert a new vertex or node in the F heap.
   public boolean insertVertexT(FHNode node)

//Performdecrease key operationson the F-Heap
    public void decreaseKey(int dkdata, FHNode vertex)
```

<u>Step 3: Calculation of Hop Sequences.</u>

Basically when traversing on a shortest path between any two vertices the first step or the first node that we come across this shortest path is termed as a hop. So the main purpose for this module is to compute a hop table having attributes source , destination and hop for each and every vertices in the input files for further calculation.
The main class that we have used for this module are hops.java which provides a standard structure for a hop with default value for a hop set to null.

Further we have declare an array list hopstable :
ArrayList<hops> hopstable = new ArrayList<hops>();

This is initially initialized for vertex to every destination for that corresponding vertex and we also store the hop values for each path. Each of these array list for all the vertices in the input files is then further stored in another array list masterhop which keeps the hop table for each of individual vertices in the input file.

ArrayList<ArrayList<hops>> vertexHopList = new ArrayList<ArrayList<hops>>();

Along with this we also need to create an underlying data structure for binary trie. The basic underlying functionality here was to create an underlying tree structure for efficient search of hops attribute when the source and destination is given. Each of the tree will correspond to hopstable for a particular vertex. We have created two main classes for this purpose IPNode and IPStructure for tree implementation. The IPNode again correspond to the attribute , constructors and method for node level operation while the class IPStructure  correspond different operations to be performed.

```
//Method to insert node in the structure
        public IPNode insertelement(Double ip,int hop)

//Method to perform preorder traversal
        public void preorder(IPNode n)

//Method to lookup an element of the structure
        public int lookupElem(IPNode root,Double ip)
```

## Step 4: Implement a routing scheme

The main concept here is to find out the routing path through use of the hops. So initially given a source vertex (IP) and destination IP we will calculate the first hop in the structure corresponding to this source vertex. Once we find the hop for the destination IP, the hop will become the new source and we will again find a new hop corresponding to this new source and the same destination IP. This process will terminate when actually the source and destination will be the same and the hop value for the same will be equal to null.

This is handled with a while loop in routing.java class file.