# COP 5536 SPRING 2015

## Advanced Data Structures
## Programming Project

Submitted By:

Sumith Chandra Reddy Nimmala

UFID: 84845046

## Objective

Part1: To implement Dijkstra's Single Source Shortest Path algorithm for undirected graphs using Fibonacci heaps.

Part2: To implement a routing scheme for a network by longest prefix matching using a binary trie. Binary Tries are computed by finding the shortest path using part1 of the project.

## Language used for the project

Java

## Compiler used for the project

Javac – Java Programming language compiler.
JDK Version: JDK 7

## Classes for Part 1

ssp.java
Node.java
AdjacencyDetails.java
fHeapNode.java
fibonacciHeap.java

## Classes for Part 2

Trie.java
TrieNode.java
Routing.java

## Structure and Prototype of classes for part 1

1. **ssp.java**

   This class contains the main method for the part1. Main method just reads the input parameters and calls the Dijkistra method to find the shortest path.

   Methods:

   1) static ArrayList<Integer> Dijkistra(String filePath, int source_node, int destination_node)(){}

      This method takes the filepath, startnode and endnode for the shortest path to be computed. This methods reads the file and creates the Nodes and then calls the ComputeCost function to calculate the shortest path. It returns the shortest path in a list to the caller function.

   2) static int ComputeCost(Node start_node, Node end_node, int no_nodes)(){}

      This method takes start_node, end_node and number_of_nodes as function parameters. Creates the fHeapNodes for each node created in Dijkistra() and runs the procedure to calculate the shortest path (dijkistra algorithm), methods like extractMin() , decreaseKey() for fibonacciHeap are called inside this method. Returns the total cost for the entire path.

2. **Node.java**

   This class maintains the below details for each node.

```
int nodeId;
int nodeValue;
ArrayList<AdjacencyDetails> adjacencyList
```

3. **AdjacencyDetails.java**

   This class maintains the below details for each edge in the graph, this class is used inside the Node.java

```
int destination;
int weight;
```

4. **fHeapNode.java**

   This class contains the Node structure required for the fibonacciHeap. Node.java class is used here to get the details of each node built previously. This class maintains the below details.

```
int degree ;
Node node;
fHeapNode child;
fHeapNode parent;
fHeapNode left,right;
boolean childCut;
```

5. **fibonacciHeap.java**

   This class implements the functionality required for the Fibonacci heap. It maintains the root of the heap which is nothing but the min value fHeapNode in our implementation.

   Methods:

   1) fHeapNode extractMin(){}
      This method removes the root of the heap and returns the new root for the heap. PairwiseCombine() is called before returning the new root.
   2) fHeapNode pairwiseCombine(){}
      This method does pairwise combine of the same degree structures in the top layer of the heap at the same time maintains the min property of the heap and returns the new root of the entire heap to the caller function.
   3) void removeFromList(fHeapNode rNode){}
      Takes the fHeapNode to be removed as a parameter and removes the fHeapNode from a particular list and updates the pointers of the circular list.
   4) fHeapNode decreaseKey(fHeapNode Node, int newValue){}
      Takes the fHeapNode and the newValue of the node as parameters and implements the decreaseKey functionality of the Fibonacci heap.
      If a particular fHeapNode is removed from the heap then casecadeCut(parent) is called to adjust the heap structure as per the fibonacci heap rules and the removed node is inserted to the top layer of the heap. Returns the new root of the heap to the caller function.

5) void cascadeCut(fHeapNode parent){}
   Takes the parent of the removed node from the decreaseKey() as input. This method maintains the cascadeCut functionality of the heap. It checks and updates the heap structure by using the childcut values of the parent nodes.
6) fHeapNode Insert(fHeapNode node){}
   This method takes the node to be inserted as input and inserts it into the top layer of the heap at the same time maintains the minimum property of the heap thus returns the root of the heap.

**Structure for Part 1:**

This is the rough structure for part 1 if the project

```
public class ssp {
     main(){
          Dijkistra();
     }

     Dijkistra()[
          ComputeCost();
     }

     ComputeCost(){
          fibonacciHeap is created by calling
          for(all nodes in the graph){
               fibonacciHeap.insert();
          }

          While(not destination node){
               fibonacciHeap.extractMin();
               for(each in adjacency list of min node){
               maintain path & call decreaseKey if required
                    fibonacciHeap.decreaseKey();
               }
          }
     }

}
```

## Structure and Prototype of classes for part 2

1. **routing.java**

   This class contains the main method for the part2. Main method just reads the input parameters and calls the RoutingScheme method.

   Methods:

   1) static void RoutingScheme(String filePath, String filePathIP,int source_node, int destination_node)(){}

      This method takes the filepath1,filePath2, startnode and endnode as parameters. This method reads the ipfile and calls the Dijkistra function of ssp.java to calculate the shortest path.

      Once shortest path is obtained, for each node in the shortest path ComputeCost() of ssp.java is called taking each node in the graph as destination and then trie is built for each node in the shortest path based on the output of ComputeCost().

      After each Trie is built pruning() is called. Once all the tries are obtained we call the findpath() for each node in the shortest path.

   2) static String convertIpToBinaryString(String ip){}

      To convert given IP to 32 bit binary string.

2. **TrieNode.java**

   This class maintains the below details for each node.

```
boolean isLeaf;
int splitIndex;
TrieNode zero;
TrieNode one;
TrieNode parent;
String ipValue;
int nextHop;
```

3. **Trie.java**

   This class implements the functionality required for the Trie. It maintains the root of the Trie.

   Methods:

   1) void insert(String IP, int nextHop){}

Takes the new Ip and the nextHop as input parameters. This method creates the TrieNode based on the input values and inserts the TrieNode into Trie by taking care of all the pointers in the TrieNode.

2) TrieNode pruning(TrieNode node){}
   This method takes the root of the Trie as an input and prunes the tree based on the nextHop values of the leafNodes.
   This is called recursively and returns the root of the trie.
3) int findSplitIndex(String ipValue, String IP) {}
   Given two 32 bit IP values, this function returns the first index at which they differ.
4) String findPath(String destIP){}
   This method is used to find the routing path of the given destIP in the trie. It returns the String based on the splitValue of the BranchNode(TrieNode) just above the leafNode(TrieNode).

**Structure for Part 2:**

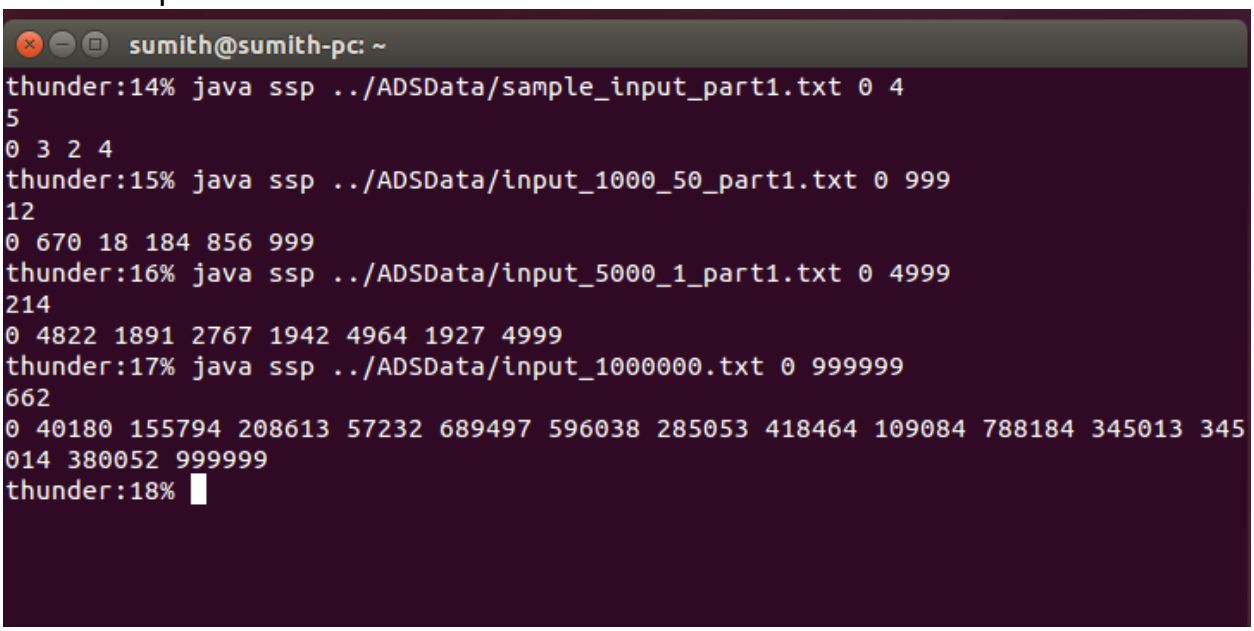
This is the rough structure for part 2 if the project

```
public class routing {

     main(){
          RoutingScheme();
     }
     RoutingScheme(){
          Ssp.Dijkistra();
          For(each node in path of Dijkistra){
           For(each node in graph){
               Ssp.ComputeCost();
               Trie.Insert();
            }
               Trie.pruning();
          }

          For(each node in path of Dijkistra){
               Trie.findPath();
               //add all the paths to a string
          }
          Print the path
     }
}
```

**Run Part 1:**

1) Go to the folder and run "make"
2) Java ssp <filepath> <start> <end>

   Result for part 1:




**Run Part 2:**

1) Go to the folder and run "make", if you have run make previously ignore this.
2) Java routing <filepath_graph> <filepath_Ip> <start> <end>

   Result for part 2: