Advanced Data Structures

Project Report

Umar Majeed
UFID: 96119336
Email: umarmajeed@ufl.edu

Contents

Objective:

This project consists of two parts, the main aim of the project is to create two Make Files, named *ssp* and *routing*.

When *ssp* is called, the project should calculate the shortest path between a source and a node given in a specific file, and display the value of the shortest path, along with the nodes that are visited to get to the destination from the source following the shortest path. The Dijkstra's shortest path algorithm is supposed to use Fibonacci heaps.

When *routing* is called, it should display the shortest path between the source and the node, the limitation of this implementation involves adding all the next hops to a binary trie and searching for the shortest path from that trie for efficiency.

Compiler and Version:

The language used for the implementation of this program is JAVA. The program was developed and tested using JAVA in Eclipse IDE in windows environment. The MakeFile.txt written uses the JAVA compiler of any OS to compile the JAVA files. The version of JAVA used in this project is JRE 1.8.0_31.

Steps for Execution:

The zip folder contains all the java classes (explained later) which are needed to run the project. Since the project involves two parts, two main classes, named *ssp* and *routing* are created.

Dijkstra's Algorithm:

To find the shortest path in a graph, use the following command:

java ssp filename source destination

The program extracts the arguments from the command, opens the file to create a graph using the adjacency list, and then runs the Dijkstra's algorithm (using Fibonacci Heaps) to find the shortest distance between the source and all nodes. The program then displays a path between the source and the destination node.

Router Next Hops:

To find the next hops of routers, run the following command:

java routing filename1 filename 2 source destination

The program extracts the arguments from the command, opens the file to create a graph using the adjacency list, and then runs the Dijkstra's algorithm (using Fibonacci Heaps) to find the distance between the source and all nodes. The program then creates a binary trie of next hops

of all the router paths from all the routers. At last the search traversal is done and the next hop routers for the destination are found and the respective binary tries are searched. This kind of traversal decreases the whole 32 bit string match and the searching becomes more efficient.

Function Prototypes:

Edge Class:

The edge class defines the *to* and *weight* variables including their getter and setters, this class defines an edge that may exist between two nodes.

Graph Class:

This method will create a graph from the input file using values extracted in File handling (Explained Later):

```
public Graph(int vertices, int edges)
```

Since it is an undirected graph, two edges are drawn for each edge, this task is handled by the following function:

```
public void DrawEdge(int e1, int e2, int w)
```

FileHandling Class:

This procedure will take a file, and read it, extracting the edges and vertices to create and return a graph:

```
public static Graph CreateGraph(Path fileName) throws IOException
```

Function to convert the IP's into 32 bit binary number:

```
public static List<String> ConvertToBinary(Path fileName) throws IOException
```

FibonacciHeap Class:

This Class makes use of the FibNode Class, which defines a node of the Fibonacci Heap.

The Shortest Path algorithm will take create a Fibonacci heap and delete the minimum items from it to update the shortest paths, for that, we need an extract min procedure, which is defined by:

```
public FibNode dequeueMin()
```

This method will merges two fib heaps in linked lists, the method is used in delete min procedure and when there is a cascading cut:

```
private static  FibNode mergeLists(FibNode one, FibNode two)
```

This method is used to decrease the key. If there is a shorter distance than the one already calculated, the algorithm will use the decrease key operation to decrease the distance of a specific node from the source:

```
public void decreaseKey(FibNode node, int newPriority)
```

To implement the cascading cut, we have another procedure named:

```
private void cutNode(FibNode node)
```

DijkstraFibonacci Class:

This method takes a graph parameter and sets variables to calculate the shortest distance:

```
public DijkstraFibonacci(Graph g, int source, int destination)
```

This method will calculate the shortest distance from a source to all the nodes using Fibonacci heaps. The function will start from the source node and keep on enqueing new nodes as it traverses more children. It is also going to traverse all nodes and calculate the updated paths from each node. The final result will be stored in the node variable.

```
public void CalculateShortestDistance()
```

This function Displays the shortest path to the destination node.

```
public void DisplayShortestPaths()
```

BinaryTrie Class:

This class makes use of the TrieNode class, which essentially defines a node of the binary trie, the node can be of two types, Branch node or an element node (the enum *NodeType* defines both the node types). The branch node will be initialized by no data where as an element node will be initialized by some content.

This method inserts a routing table of the next hops into the binary trie array variable (which is a trie of arrays). The routing table contains the destination node and the next hop calculated from the Fibonacci Dijkstra. We are essentially constructing the whole trie in a single function here:

Format: routingTable[number of IP addresses][2]

```
public void InsertRoute(String[][] routingTable)
```

The cleanup process to move the nodes with the same address one level up. It's a recursive procedure which checks the element nodes and moves them up by comparing the element nodes with its siblings:

```
private void Cleanup(TrieNode node)
```

The search function works like the binary search instead that it returns the next hop that needs to be searched for the destination node:

```
public String SearchNextHop(String destAddress)
```

IpRouting Class:

This class will take the input graph and the input ip list to generate a routing table and pass it on to the trie to generate a trie of next hops.

This procedure initializes the Ip routing class and calls the create routing table function:

```
public IpRouting(Graph g, List<String> ipList, int source, int destination)
```

This function will calculate the shortest path from each source node to each node, and then create a binary trie using the shortest paths created:

```
private void CreateRoutingTable()
```

to find the shortest path, the next hop is searched in source binary trie and then the search is moved on to the next hop's binary trie:

```
public void FindPath()
```

Program Structure: