Advances Data Structures (COP 5536)

Spring 2015

Programming Project Report

Juthika Das

UFID 7173-5283

juthika@ufl.edu

PROJECT DESCRIPTION

The project consists of two parts. The first part is implementing the Dijkstra's algorithm using Fibonacci Heaps. Dijkstra's algorithm is used to calculate the shortest path between a given source to a given destination. In the first part of the project, we read a file from the command line to create an adjacency matrix.This is in the form of an undirected graph. Now, we run Dijkstra's algorithm on it and keep updating the value of the adjacency matrix until the destination is reached.

In the second part of the program, We're required to build tries at each router. A trie is based on the IP Address of the router. At the end of the trie, we store the next hop of the first IP Address in the path to the last IP Address. We then remove the subtries which have the same next hops. The trie is finally traversed from the destination to the next hop node.

WORKING ENVIRONMENT

HARDWARE REQUIREMENT

Hard Disk space: 4 GB minimum

Memory: 512 MB

CPU: x86

OPERATING SYSTEM

Windows 8.1

COMPILER

Javac

COMPILING INSTRUCTIONS

The project has been compiled and tested on thunder.cise.ufl.edu and Eclipse Juno.

To execute the program,

You can remotely access the server using ssh username@thunder.cise.ufl.edu

For running Dijkstra's algorithm on an input file, type

Java ssp 'file path' source destination

I've included the file million.txt that has 1M nodes.

So to run Dijkstra's on that, type

Java ssp million.txt 0 999999

For running the second part, type

Java routing ‘graph path’ ‘IP Address file path’ source and destination

I’ve included the files ‘input_graphsmall_part2.txt’ and ‘input_ipsmall_part2.txt’

So, an example using these files would de

Java routing input_graphsmall_part2.txt input_ipsmall_part2.txt 1 6

STRUCTURE OF THE PROGRAM AND FUNCTION DESCRIPTIONS

There are 7 classes that I have used to implement the programming assignment. Out of those 7 classes, ‘Fibonacci.java’ , ‘FibonacciHeapNode.java’ , ‘Graph.java’ and ‘SSP.java’ are used for implementing the first part of the project. The programs ‘Tries.java’ , ‘Routing.java’ , ‘Arrange.java’ are used to implement the second part while also using the first 4 programs since we use Dijkstra’s algorithm to complete the second part of the project.

Here’s how the functions are defined in each class and a short description of how they work.

FibonacciHeapNode.java

The class FibonacciHeapNode mainly describes the structure of a node that is used in a Fibonacci heap.

The class variables are:

Degree

This variable is of type Integer and it signifies the number of nodes that a node can have in its next level.

childCut

This variable is of type Boolean and it signifies whether or not a child has already been removed from that node. A childCut of false means that no child has ever been removed from that node.

Key and index

Key stores the value of the Fibonacci node whereas index stores the index signifies the number of the Fibonacci node.

The fibonacciHeapNode class further has objects parent, child, right and left. Right and left are used for the doubly linked list that is created at each level.

The functions of this class are:

Int keyValue()

Return type: Integer

Parameters: Null

This function returns the value of the Fibonacci Heap node.

Fibonacci.java

The class Fibonacci is where all the operations of a Fibonacci heap have been defined. The operations of this class are performed on instances of class FibonacciHeapNode.

The functions defined in this class are:

Void insert(FibonacciHeapNode myNode, int key, int index)

Return type: void

Parameters : fibonacciHeapNode myNode, int key, int index

This function inserts a node into the Fibonacci heap. The function works in two ways. If the min is null i.e if the root is null, myNode is assigned to be the root of the Fibonacci heap since it is the only root in the Fibonacci heap right now. However, if there is a min root, myNode is added to the top level of the Fibonacci heap, to the right of the min root in the doubly linked list of that level.

FibonacciHeapNode removeMin()

Return type: FibonacciHeapNode

Parameters: None

This function removes the minimum node, min, from the Fibonacci heap. Since the min is the root of the Fibonacci heap, on removing the root, we'll have to adjust the nodes on the next level of the root. So, a variable 'z' stores the degree of the root and for each child, we place the child subtree on the topmost level until z is 0. The min node is then returned. This class calls the meld() function to meld the subtrees with the same number of nodes, at the top most level.

Void meld()

Return type: void

Parameters: none

This functions checks which nodes at the topmost level have the same degree. If there are two nodes with the same degree, it melds the two subtrees by attaching the subtree with larger root value as the child of the subtree of the smaller tree value and adjust min.

Protected void removeNode(FibonacciHeapNode remChild, FibonacciHeapNode degNode)

Return type: void

Parameters: FibonacciHeapNode remChild, FibonacciHeapNode degNode

This functions performs the task of removing a node from its doubly linked list. If remChild is a child of the node degNode, then the node to the right of remChild is set as child of degNode. It is then added to the top most level to the right of min.

Public void link(FibonacciHeapNode degNode , FibonacciHeapNode remChild)

Return type: void

Parameters: FibonacciHeapNode degNode , FibonacciHeapNode remChild

This function is used to add the link that we talked about in the removeNode function. We're adding degNode as a child of remChild and this function makes all the necessary changes in the doubly linked list.

Public void decreaseKey(FibonacciHeapNode remChild, int decVal)

Return type: void

Parameters: FibonacciHeapNode remChild, int decVal

This function decreases the value of remChild to decVal, that is passed as an argument.

If the value of remChild goes below the value of its parent, we need to remove remChild using the removeNode() function and use the cascadingCut() function the check the child cut values of the parent and the ancestors.

Public void cascadingCut(FibonacciHeapNode degNode)

Return type: void

Parameters: FibonacciHeapNode degNode

This function checks the childcut values and makes the necessary changes and performs remove if needed.If the childcut value is false for the parent, make it true. If the childcut is true, keep going up the tree and removing the nodes until it finds a node whose childcut is false.

Graph.java

The class Graph encapsulates another public class DefGraph which lends the graph its properties such as index and weight. The class DefGraph defines functions:

Public void display()

Return type: void

Parameters: none

This function displays the index and the weight.

Public void getInd() , public void getWt()

These functions get the index and weight of the entity respectively.

The class Graph has the following functions defined in it:

Public void displayGraph()

Return type: void

Parameters: None

This function displays all the edges and vertices of the graph.

Public void addToGraph(int v1, int v2, int w)

Return type: void

Parameters: int v1, int v2, int w

This function adds nodes to the adjacency list with the weight of the edge.

public LinkedList<Integer> dijkstra(int sourceNode, int desNode)

Return type: Linked List of integers

Parameters: SourceNode and desNode

This function runs the Dijkstra's algorithm on the graph using the Fibonacci heap functions such as insert(), removeMin() and decreaseKey().

Public HashMap<Integer, LinkedList<Integer>> dijkstra(int SourceNode)

Return type: Hashmap with key of type Integer and LinkedList of Integers as the value type.

Parameters: int sourceNode

This function inserts nodes in Fibonacci heap using the insert() function. After inserting the nodes per level from the graph, it performs a removeMin() to get the shortest path from that node in the graph. After rearranging the Fibonacci heap, the same process is followed. It involves desreaseKey() if we find a path that is shorter than the path mentioned in the graph. The value is then updated in the graph.

ssp.java

This class contains the main function that runs the Dijkstra's algorithm on the graph.

The functions are:

Public static Graph readGraph(String path)

Return type: Graph

Parameters: String path

This function is responsible for reading the input file and returning a graph with edges and vertices as mentioned in the file. Using an array named worksets[], we store the values of vertex1 (v1) , vertex2 (v2) and the weight, from the input file. Using the addToGraph() function, we add these values to the graph and return the graph.

The **main** function takes the source Node and destination Node for Dijkstra's algorithm and runds the dijkstra() function of them.

This ends the function descriptions for the first part of the project, that is, dijkstra's using Fibonacci heaps.

For the second part of the project, the following classes and functions have been used.

Arrange.java

Public static Graph makeGraph(String location)

Return type: Graph

Parameter: String location of the input file

This function simply reads the input file and creates a graph out of it. It reads the vertices and the edge weight between the vertices and adds this information as a graph edge between the two vertices.

Public static char[] intToBinary(String inputData)

Return type: char[]

Parameters: String inputData

Since we'll be using tries, the IP Addresses need to be converted to binary and so this simple function calculates the binary equivalent of the IP Address.

public static LinkedList<char[]> getIP(String location, int numberNodes)

Return type: Linked List of character arrays which store the IP Addresses

Parameters: String location, int numberNodes

This function reads the IP Addresses from the files and converts it to binary using the intToBinary() function defined above.

Tries.java

The Tries class is used to defines the operations of tries and the postorder traversal for pruning the trie at each router.

This contains the private class TrieNode. The TrieNode class describes the properties of the node in a trie such as its parent, zeroChild that is the left child and oneChild that is the right child.

The variable Boolean branchCheck returns true if the Node is a branch node. There are two constructors of class TrieNode that initialize the class variables.

The constructor of class Tries, Tries() sets the root as null.

Void insert(char[] dest , int nextHop)

Return type: void

Parameters: destination IP and the nextHop

This function inserts the destination IP and the nextHop in the trie. If the trie root is set to null that is, if the trie is empty, then the dest[] is added as the trie root.

While the node is a branch node, keep adding the bit of the dest[] as a zeroChild if it is a 0 or a oneChild if it is a 1.

The function then goes on to find the first different bit and creates a branch there using the branchAt() function.

private void branchAt(char[] dest, int nexthop, TrieNode branchNode, int pos)

Return type: void

Parameters: char[] dest, int nexthop, TrieNode branchNode, int pos

This function creates a branch node where a bit difference is encountered. It traverses up the trie till it finds a branch node that has branch index position less than pos. It then creates a new branch node that has branch index position = pos, adds it as left/right child to the branch found above. Then adds the new leaf node and old node (that was previously at the position of the new branch node),as its children

Incase branchNode becomes null while traversing up,store the old root in a temp variable, create a new root branch node with branching index = pos, add the new leaf node and the old root as it child and make this new branch node as root.

Public void minimize(TrieNode node)

Return type: void

Parameters: TrieNode node

This function compresses the Trie by checking for unused branches and marking the character array with '*'.

Public char[] search(char[] destination)

Return type: char[]

Parameters: char[] destination

This function goes through the trie to look for the destination by moving left or right depending upon the value of the IP at a given index.

routing.java

The Routing class contains the main function for the second part of the project.

public static Graph readGraphFromFile(String path)

Return type: Graph

Parameters: String path

This method takes the input file and reads through it to make a graph. For each line in the input file, it separates the values of v1, v2 and the weight and then adds this edge starting and ending at v1 and v2 and with weight w to the graph.

public static LinkedList<char[]> getIP(String location, int numberNodes)

Return type: Linked List of character arrays which store the IP Addresses

Parameters: String location, int numberNodes

This function reads the IP Addresses from the files and converts it to binary using the intToBinary() function defined above.

Public static char[] intToBinary(String inputData)

Return type: char[]

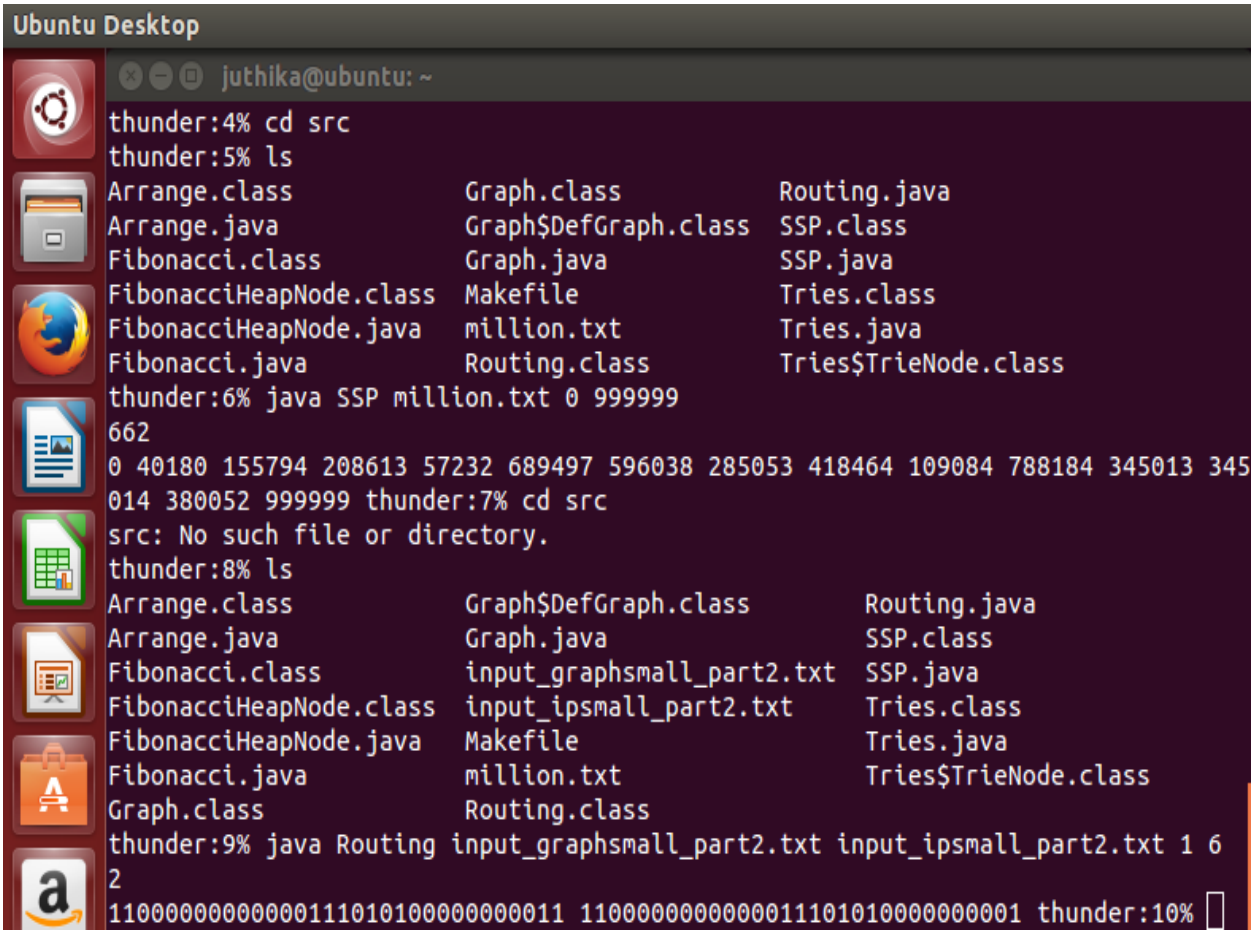
Parameters: String inputData

Since we'll be using tries, the IP Addresses need to be converted to binary and so this simple function calculates the binary equivalent of the IP Address.

The main() function reads the graph file, the IP file, the source and the destination. It runs Dijkstra on the graph. We then use the insert() function to enter the IP Addresses into the tries. We then branch the tries and compress them using the minimize() function.

The time taken to run ssp on 1M nodes was found out to be 128869 ms.

RUNNING THE PROGRAMS



CONCLUSION

The first part, that is, implementing Dijkstra's algorithm using Fibonacci heaps has been successfully implemented within acceptable time limits for the given input files. The second part has been further successfully implemented using the Dijkstra's shortest path.