Assignment 4

CS5310

Jonah Kubath

Nov 9, 2018

Assignment 4 is using Kruskal’s algorithm to determine the minimum cost spanning tree. The algorithm begins by building a min-heap with all the of edge weights. The root of this heap is removed each iteration. The nodes on either end are checked to determine if combining the two will create a cycle. This is done by updating the parents of each node. If the two nodes are part of a tree with the same root, then adding the edge will create a cycle. This check is also improved by using the collapsing find function which resets the root values to limit the number of iterations required to determine the root of a tree. The algorithm is iterated until n–1 edges are used in the path or the min-heap is out of edges.

I tested by my program by running to sets of tests. The first was on sets of data from 10 – 100 nodes. The second test was 100 – 1000 nodes. The input files were generated by using the Random class in Java. Two for loops were used to iterate over each possible path between the nodes. Random.nextBoolean() would decide if a path was to be generated. Random.nextInt() was then used to generate a random path weight. In my testing, I set the range of path weights from 0 – 100.

The output of my program shows the execution time increases slightly more than linearly as the number of nodes is increased.

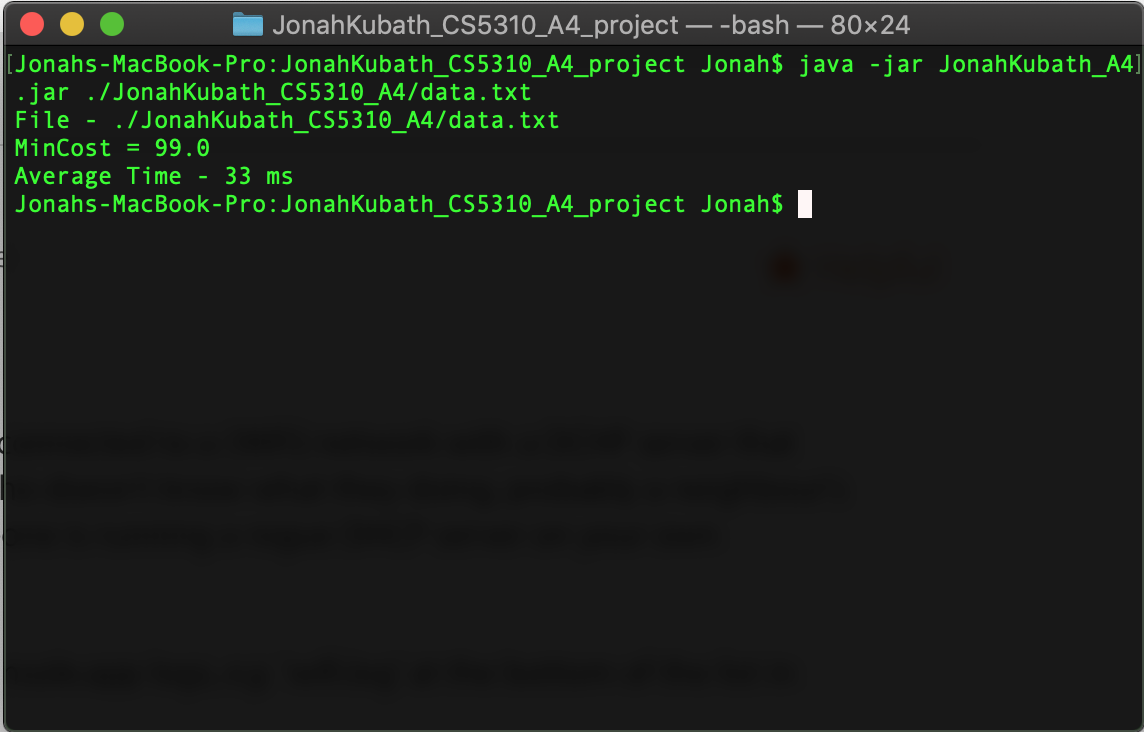
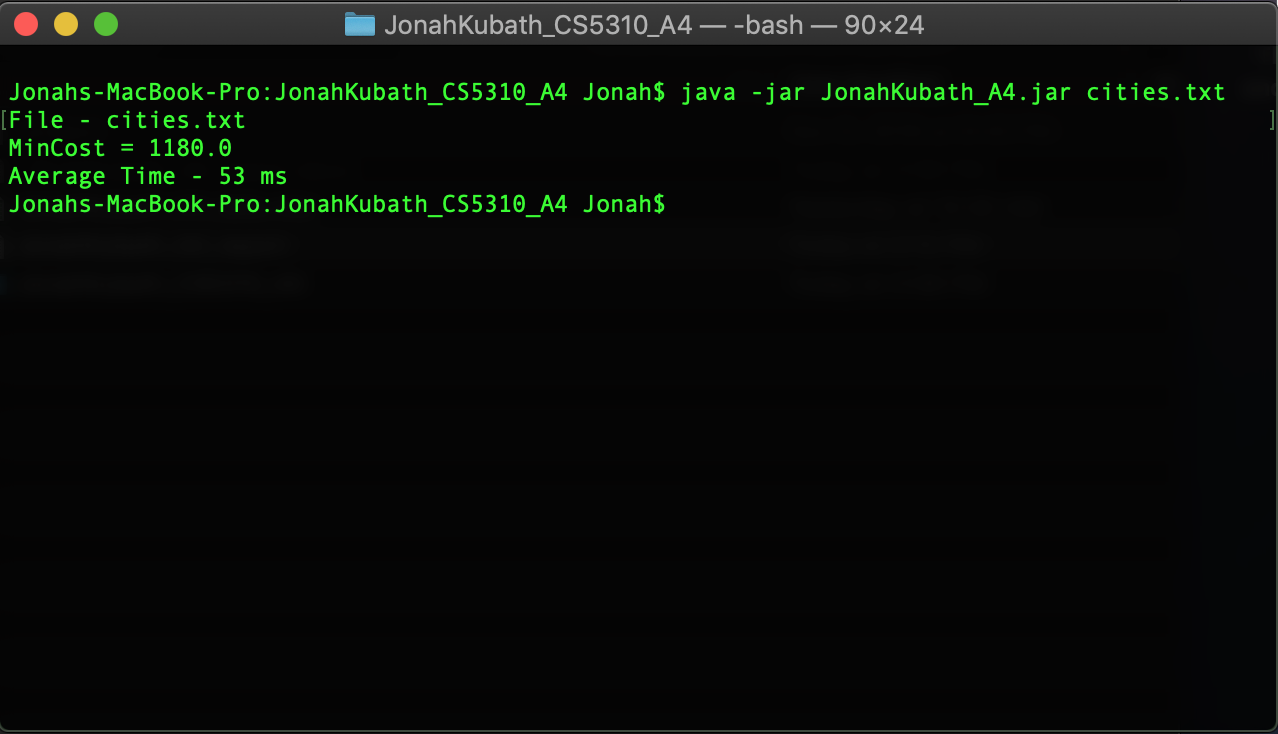
Execution:

I used Eclipse to load my project folder and run the program for the test files.

I have included a runnable jar file. This can be run on the command line in the Java JVM. Navigate to the project directory that includes the .jar file.

java -jar JonahKubath\_A4.jar <filename>

<filename> must be changed to the location to the input file. “./JonahKubath\_CS5310\_A4/data.txt” is a possibility.



package jonahKubath\_A4;

import java.io.File;

import java.io.FileNotFoundException;

import java.util.Iterator;

import java.util.PriorityQueue;

import java.util.Scanner;

public class Main {

static int[][] cost;

static int[] parent;

static PriorityQueue<Edge> heap = new PriorityQueue<Edge>(new SortEdge());

static int[][] path;

static boolean isDirected = false;

public static void main(String[] args) {

String f = "./data\_10\_100/data\_0.txt";

//filename passed as argument

if(args.length > 0) {

f = args[0];

}

System.out.println("File - " + f);

int runs = 5;

long totalTime = 0;

double minCost = 0;

for(int i = 0; i < runs; i++) {

readData(f);

//printCost(cost);

//printHeap(heap);

Long start = System.nanoTime();

minCost = Kruskal(heap, cost, cost.length);

Long end = System.nanoTime();

totalTime += (end - start);

}

System.out.println("MinCost = " + minCost);

System.out.println("Average Time - " + totalTime / runs / 1000 + " ms");

}

/\*\*

\* Kruskal's algorithm for min cost tree

\* @param e Heap of Edges

\* @param cost Cost array

\* @param n Number of nodes

\* @return MinCost as a double

\*/

public static double Kruskal(PriorityQueue<Edge> e, int[][] cost, int n) {

int i = 0;

double minCost = 0;

Edge cur = null;

int j = -1; //Parent of left node of an edge

int k = -1; //Parent of right node of an edge

while((i < n - 1) && e.size() > 0) {

cur = e.remove();

j = Find(cur.getL());

k = Find(cur.getR());

if(j != k) {

//System.out.println((cur.getR() + 1) + " " + (cur.getL() + 1));

i++;

path[i][0] = cur.getL();

path[i][1] = cur.getR();

minCost += cur.getCost();

Union(j, k);

}

}

if(i != (n-1)) {

System.out.println("No Spanning Tree");

return 0.0;

}

else

return minCost;

}

/\*\*

\* Collapsing find function

\* @param i node to find the parent of

\* @return The index of the root

\*/

public static int Find(int i) {

int cur = i;

while(parent[cur] >= 0) {

cur = parent[cur];

}

/\* Collapsing \*/

int index = i;

int temp = 0;

while(index != cur) {

temp = parent[index];

parent[index] = cur;

index = temp;

}

return cur;

}

/\*\*

\* Union two trees of nodes. The smaller tree is added to the bigger tree.

\* @param j Index of the left root

\* @param k INdex of the right root

\*/

public static void Union(int j, int k) {

//K has more nodes

if(k < j) {

if(parent[j] > 0)

parent[k] += (parent[j] \* -1);

else

parent[k] += parent[j];

parent[j] = k;

}

else {

if(parent[k] > 0)

parent[j] += (parent[k] \* -1);

else

parent[j] += parent[k];

//parent[j] += parent[k];

parent[k] = j;

}

}

/\*\*

\* Read the data from the given file. If the file is not found, the user is prompted for a new name.

\* @param filename String name of the file

\*/

public static void readData(String filename) {

Scanner scan = null;

String file = filename;

File f = new File(file);

try {

scan = new Scanner(f);

}

catch(FileNotFoundException e) {

System.out.println("File not found");

System.exit(0);

}

//Read the number of nodes

int number = Integer.parseInt(scan.nextLine());

//Read the start position

String line[];

cost = new int[number][number];

parent = new int[number];

path = new int[number][2];

if(isDirected) {

for(int i = 0 ; i < number; i++) {

//Split and leave empty spots as blank

line = scan.nextLine().split(",", -1);

//Save the line

for(int j = 0; j < number; j++) {

if(line[j].compareTo("") != 0) {

cost[i][j] = Integer.parseInt(line[j]);

//Add the edge to the MinHeap of edges

if(cost[i][j] != 0) {

Edge e = new Edge(i, j, cost[i][j]);

heap.add(e);

}

}

else

cost[i][j] = -1;

}

parent[i] = -1;

}

}

else {

for(int i = 0 ; i < number; i++) {

//Split and leave empty spots as blank

line = scan.nextLine().split(",", -1);

//Save the line

for(int j = 0; j < i; j++) {

if(line[j].compareTo("") != 0) {

cost[i][j] = Integer.parseInt(line[j]);

//Add the edge to the MinHeap of edges

if(cost[i][j] != 0) {

Edge e = new Edge(i, j, cost[i][j]);

heap.add(e);

}

}

else

cost[i][j] = -1;

}

parent[i] = -1;

}

}

scan.close();

}

/\*\*

\* Print the cost array

\* @param map The cost matrix to print

\*/

public static void printCost(int[][] map) {

System.out.println("\nCost Matrix");

for(int i = 0 ; i < map.length; i++) {

for(int j = 0; j < map.length; j++) {

System.out.printf("%3d ", map[i][j]);

}

System.out.println();

}

}

/\*\*

\* Print the heap by duplicating the given heap and then removing the edges

\* from the newly created heap.

\* @param h The heap to print

\*/

public static void printHeap(PriorityQueue<Edge> h) {

System.out.println("\nMin Heap by edge cost");

Iterator<Edge> i = h.iterator();

Edge e = null;

PriorityQueue<Edge> newHeap = new PriorityQueue<Edge>(new SortEdge());

while(i.hasNext()) {

e = i.next();

Edge temp = new Edge(e.getL(), e.getR(), e.getCost());

newHeap.add(temp);

}

Edge cur = null;

while(!newHeap.isEmpty()) {

cur = newHeap.remove();

System.out.println(cur.getL() + " " + cur.getR() + " - " + cur.getCost());

}

System.out.println();

}

}