cs

Christine Grosso

cgrosso@bu.edu

Abstract

Two greedy algorithms are used to determine a “shortest path,” given a matrix of city-to-city distances, as well as a list of direct distances to the destination.

cs526 final project

Heuristic methods for finding a shortest path

Contents

[Introduction 1](#_Toc38653521)

[Major Data Structures used in program: 3](#_Toc38653522)

[Objects: 3](#_Toc38653523)

[Other structures: 3](#_Toc38653524)

[Pseudocode: 5](#_Toc38653525)

[High-level pseudocode of main() method: 5](#_Toc38653526)

[Pseudocode for findPath() method: 6](#_Toc38653527)

[Appendix A 8](#_Toc38653528)

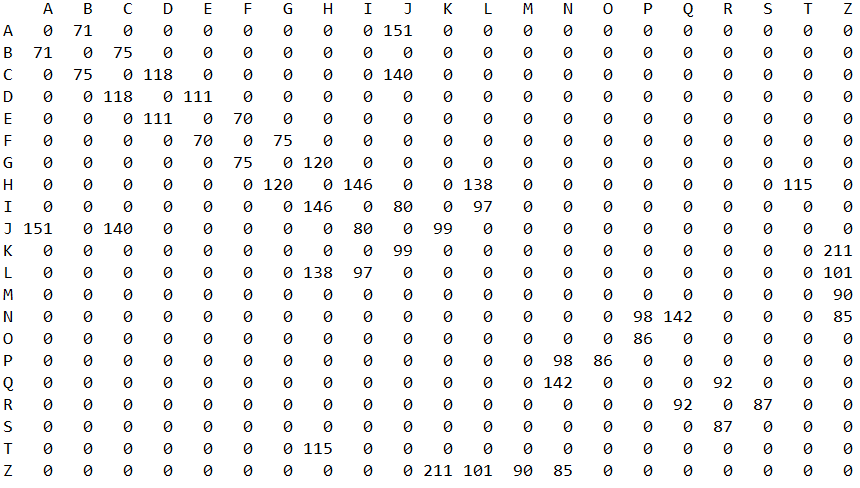
[Works Cited 9](#_Toc38653529)

# Introduction

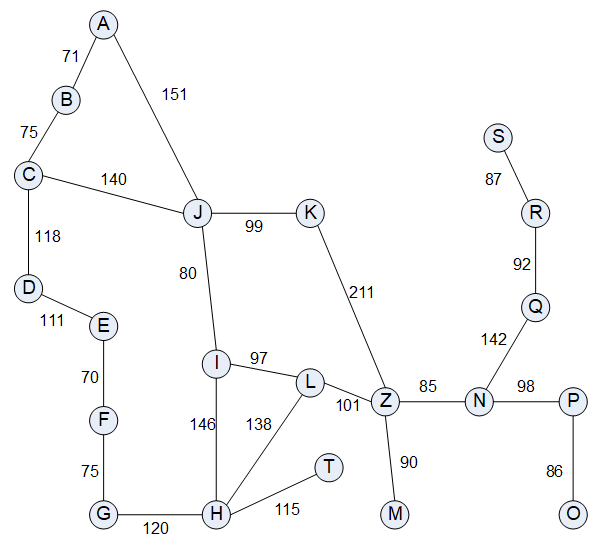
This program reads in two data files that describe distances between cities (or “nodes”) in a graph, requests user input to determine a starting node, and then determines a “shortest path” to get from the starting node to the destination node (which will always be node ‘Z’ in this program).

The first data file read in contains a “matrix” of nodes. The first line (or row) of this file contains column labels (a capital letter) which represent destination cities for the remaining rows, each of which begin with a letter representing a starting city, and integers representing a distance to get to the specific destination city in the corresponding column. This file, then, is a look-up table. Here is the main sample file used to code and test the program: (Lee PhD, 2020)

graph\_input.txt:

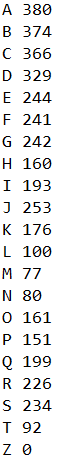


Here is a graphical depiction of the above data: (Lee PhD, 2020)

****

The second data file read in contains a list of “direct distances.” Each line in the file has only two pieces of information: a destination city node/name (corresponding to the same capital letters as above) and an integer representing the direct distance of that city to Z, if one were to be able to get to Z “as the crow flies.” Here is the file of direct distances that goes with the matrix above: (Lee PhD, 2020)

direct\_distance.txt:



Two heuristic algorithms are used to determine a “shortest path” to node Z. Both algorithms are greedy algorithms, in that they look for an optimal “local” solution, and not necessarily one that will result globally in THE shortest path. The algorithms, which will be enumerated further later, work like this:

When choosing the next node from the current node:

1. Look at all adjacent nodes that are not already on the path and choose the one with the smallest direct distance.
2. Look at all adjacent nodes that are not already on the path and choose the one with the combined (direct distance PLUS distance away from current node) that is the smallest.

# Major Data Structures used in program:

## Objects:

class GraphNode // instantiated and read in from graph\_input.txt

char name // letter representing a city/node

int weight // mileage between cities; intersection in matrix of row, column

(methods include getters and a toString method)

class DistanceNode // instantiated and read in from direct\_distance.txt

char name // letter representing a city/node

int dd // mileage from this city/node directly to ‘Z’

boolean isOnPath // indicates “visited” status; initialized to “false”

(methods include getters, a setter for the boolean, and a toString method)

class KeyValue<K, V> // generic class to allow different greedy algorithms to use the calculated mileage as a KEY and the GraphNode object as the value. This class is instantiated and used in the pathList ArrayList by the findNode() method, which will use a Comparator object to compare KEY values with which to sort the pathList ArrayList.

K k // K type will be integer. k is the mileage used as key

V v // V type will be GraphNode object as the value portion of the <K,V> pair

(methods include getters and a toString method)

## Other structures:

char[] nodeNames – array of all the node ‘names’ (alphabet letters) of the cities. Finding the name of the node in the array (done by findNode() method) yields the index to be used in the nodeMatrix as well as the dDistance Array.

GraphNode[][] nodeMatrix – 2x2 matrix containing all the GraphNode objects read in

DistanceNode[] dDistanceArray – array containing node ‘names’ and their direct distances to ‘Z’

ArrayList<GraphNode> fullPath – arrayList (used as a Queue) containing all visited nodes, including backTracks.

ArrayList<GraphNode> shortestPath – arrayList to store the nodes in the shortest path

ArrayList<KeyValue<Integer, GraphNode>> pathList – arrayList in which to collect all potential “next nodes” from the current node. After all are collected, they are sorted with a call to Collections.sort(pathList, reverseComp). The nodes (the V portion of the KeyValue node) are then pulled out from greatest to smallest, based on the K and pushed onto the S so that the smallest K is on the top of the stack.

Stack<GraphNode> S – the stack used to collect nodes that can then be popped off from best choice to less-than-best choice, especially in case of backTracking.

# Pseudocode:

This is the basic outline of how the whole program works:

## High-level pseudocode of main() method:

while user input not valid

request the name of a starting node from the user

if first time through this loop

call readGraphFile

call readDistanceFile

call validateNode

if invalid input, display message to that effect

end while

instantiate the startNode

instantiate the endNode // ‘Z’

for each algorithm

call putOnPath(startNode) // indicates that startNode is on the path

call findPath // main searching method for the shortest path

call printResults

call clearPathFlags // remove all nodes from path before next shortest path search

end for

## Pseudocode for findPath() method:

(This includes use of both heuristic algorithms. The algorithm number, 1 or 2, is passed to the method as an argument.)

This is the basic logic for the “workhorse” method of the whole program:

define fullPath ArrayList to store entire traverse // use fullPath as a Queue

define shortestPath ArrayList for shortest path

define pathList ArrayList for KeyValue pairs // see data structures section for further info

define S, a stack on which to store discovered next nodes

currentNode = startNode

while currentNode != endNode

call findNode to find the row#, r, of currentNode in the nodeMatrix

for all columns c

if nodeMatrix[r][c] != 0 // then this is nextNode VALUE for KeyValue pair

if algorithm 1 is being run

KEY = currentNode’s direct distance

else // algorithm 2 is being run

KEY = currentNode’s direct distance + distance to nextNode

instantiate new KVObject

add KVObject to pathList

end if

end for

use Comparator to sort pathList by KEY, in non-increasing order

if pathList is empty // no close cities were found—must backTrack

call backTrack()

else

while pathList !empty

remove first element (farthest away) & push onto S

end while

end else

while S !empty

pop off closest node // closest will be at the top of the Stack

add node to fullPath ArrayList

if node is not already on the path

add node to shortestPath ArrayList

break;

end while

if S is empty print message that something is wrong and exit

end while

# Appendix A

1. Note to facilitator: The Global.java interface file contains the hard-coded names of the input data files, where they can be edited there for your purposes.
2. Assumptions: The node/city names (i.e., the capital letters read initially from the graph\_input.txt file) are expected to be in the same order in the graph\_input.txt file as they are in the direct\_distance.txt file. Another student verified with Dr. Lee at the end of one of his lectures that this was an acceptable assumption.

# Works Cited

Lee PhD, J. Y. (2020, Spring 02). Assistant Professor of Computer Science. *Final Project Documentation for CS526*. Boston, MA, USA: Boston University.