CS 526: Final project documentation

Author: Ragesh Damodaran

Date: 12/10/2020

# Synopsis

This document contains details on how to execute the final project, pseudocode of the algorithms and details on data structures used.

# How to execute the code

**Prerequisites**

The GraphHelper class has constants that hold the location of the input files, these must be modified to point to the location of the actual file in your computer before running the code.

final static String *GRAPH\_INPUT\_FILE* = "/Users/ragesh/Documents/Boston/CS526/CS526\_code/Damodaran\_Ragesh\_project/resources/graph\_input.txt";  
final static String *DIRECT\_DISTANCE* = "/Users/ragesh/Documents/Boston/CS526/CS526\_code/Damodaran\_Ragesh\_project/resources/direct\_distance.txt";

**Running**

The main module that interacts with the user is the class named GraphHelper, executing the class will prompt the user for input and show the shortest distance based on both the algorithms, example:

Enter your start node letter : J

User enters node J as the start node

Algorithm 1:

Sequence of all nodes: J->K->Z

Shortest path: J->K->Z

Shortest path length: 310

Algorithm 2:

Sequence of all nodes: J->I->L->Z

Shortest path: J->I->L->Z

Shortest path length: 278

In case of an incorrect input the code will keep prompting the user for correct input , example :

Enter your start node letter : sasddas

Incorrect input, please enter an alphabet in upper case

Enter your start node letter : U

Incorrect input, please enter existing nodes

**Pseudocode**

I am using an iterative approach to traverse the nodes and use a stack to keep track of the traversal. I additionally also maintain a list of visited nodes to print at the end, this list will contain nodes we had to back track from also.

**Traversal Logic as pseudocode**

while current\_node != end\_node:

current\_node.visited = true

stack.push(current\_node)

list.add(current\_node)

distance = Integer.Max

hasVisitableChildren = false

for edge in current\_node:

if edge.destination\_node!=visited and edge.weight!=0:

if algorithm == 1:

new\_distance = edge.destination\_node.direct\_distance

if algorithm == 2:

new\_distance = edge.weight + edge.destination\_node.direct\_distance

if new\_distance< distance:

      distance = new\_distance

      next\_node = edge.destination\_node

hasVisitableChildren = true

if hasVisitableChildren == false:

stack.pop()

current\_node = stack.pop()

else

current\_node = next\_node

**Traversal logic as steps**

* Loop until the current node is not equal to the end node (i.e. Z)
* Inside the loop mark current node as visited
* Push current node to stack
* Add current node to list of visited nodes
* Loop through all edges of the current node
* For each edge, get distance calculated for each edge based on algorithm specified (1 or 2), details of algorithm are provided below
* We compare this distance from previous step with a distance variable which initialized in the outer loop as Integer.MAX and if its less that this variable we reassign the value to this and mark the node as next node
* This process is continued for all the edges of the current node
* If there are any next nodes to visit we do that and the outer loop continues until we reach the end node (Z)
* In case there are no child nodes in a node we visited, then we back track by popping the current node and taking the next node in the stack as the current node this will cause the algorithm to explore another branch of the tree

**Algorithm 1**

For each edge, get its connected node letter and for that node letter get its direct distance from the direct distance map, this is the distance for algorithm 1.

**Algorithm 2**

For each edge, get its connected node letter from the direct distance map as well as the edge weight and add these, this determines the distance for algorithm 2.

**Data Structures used**

**Custom classes**

The below are the custom classes that were created:

1. Node: The class has the members as below and is used to describe a node in the graph

char letter;  
List<Edge> edges;  
int distanceFromZ;  
boolean visited;

1. Edge: The class has the members as below and is used to describe an edge originating from a node to another node

int distance;  
char destNodeLetter;

**Java data structures**

The below java data structures are also used in the program:

1. Stack<Node> nodeStack: To hold the traversed nodes in the graph
2. List<Node> allVisitedNodeList: To hold all the traversed nodes including one’s which had to be backtracked
3. HashMap<Character, Node> charNodeMap: To hold mapping between a node letter and the node