BOSTON UNIVERSITY

Data Structures and Algorithms

CS526

Phillip Escandon

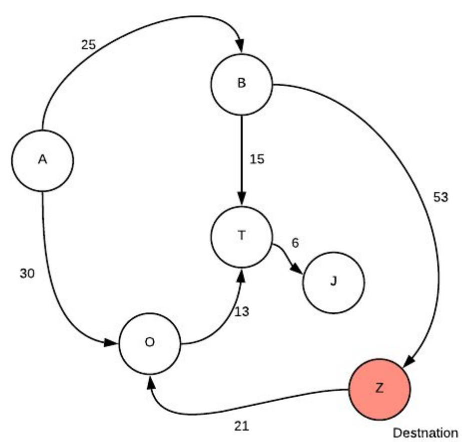
Final Project

Overview

This project required a design and implementation of two heuristic algorithms to find the shortest path in a graph. Two files were used as input.

- Graph_input.txt
  - Textual representation of the graph
- Direct_distance.txt
  - Distance from each node to the target node Z

Problem Description



From any given node find the shortest distance to the Z node. You could not visit a node more than once.

To find the shortest distances, two algorithms were written and are described below. They each had the following requirements.:

Definitions:

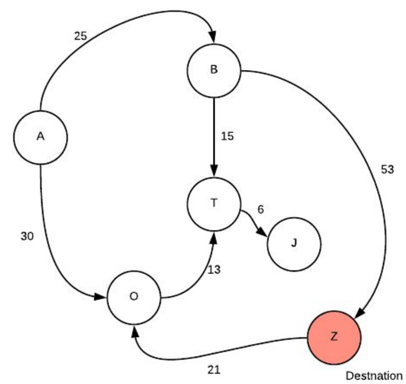
dd(v) : Direct Distance from the node to the target node.

w(n,v): Weight values of the edge between node *n* and node *v*

- Algorithm 1
  - Among all nodes *v* that are adjacent to the node *n*, choose the one with the smallest direct distance, noted as dd(v).
- Algorithm 2

    ○ Among all nodes *v* that are adjacent to the node *n*, choose the one for which the weight of the edge between nodes, designated as *w(n,v)* + the direct distance *dd(v)* is the smallest.

Approach



The main thrust of my approach was to use a nested hashmap to construct and contain the nodes and its associated edges and weights. In the textbook, this is referred to as an adjacency map and I could understand this method easiest.

Once the nodes were constructed, the path forward in implementing the two algorithms were relatively straightforward, with the exception of the back tracing which will be described later.

Once the nodes were constructed I ensured that I could complete all actions such as :

- Retrieving all nodes
- Retrieving distance between nodes
- Listing all edges for a given node
- Retrieving direct distance from a given node.
- Finding a given node

At this junction I could begin writing and debugging my Algorithm 1 and Algorithm 2 code attempts

PseudoCode

```
// Main
Instantiate p class and build nodes
Gather user input
startNode, startNode2 = input
Algorithm 1
While  startnode != Z
      Track startnode
      startNode = getShortestDD(startNode)
Print results

Algorithm2
Startnode = startnode2

While startnode != Z
      Track startnode
      startNode = getShortestDDPlusWeight(startNode)
Print results
```

Algorithm 1: `findShortestDD(startNode)`

```
Iterate over the edges associated with the startNode
      If only 1 edge this is a dead end
            Assign it as the closest node available
      else if edge node is in nodeTracer && first element in nodeTracer
            Do not assign
      else  compare currentDistance with edgeDistance
            pick the closest distance
            assign node name with the distance
Return nodeName
```

Algorithm 2: findSortestDD_Plus_Weight(node)

```
Iterate over the edges associated with the startNode
      If only 1 edge this is a dead end
            Assign it as the closest node available + WeightOfEdge
      else if edge node is in nodeTracer && first element in nodeTracer
            Do not assign
      else  compare currentDistance with edgeDistance
            pick the closest distance + WeightOfEdge
            assign node name with the distance + WeightOfEdge
Return nodeName
```

Once the first algorithm was debugged, instituting the second algorithm was trivial. The difficulty was formulating a back tracing structure to contain a list of what nodes had been

visited. I chose an arrayList called 'nodeTracer' to maintain this list. Each algorithm had its own nodeTracer array.
At the end of the algorithm the nodeTracer was filled with the complete path of nodes taken.

I then created a node cleaner that would remove the redundant sections and calculate the shortest distance. A stack implementation, I believe could have worked here as well.

Graph File Description

The graph file was read in and used to create a 2-D adjacency list array, this array would be used later to construct

1. Edges with weights
2. Nodes that contain those edges. This was handled in a for-for-if loop, resulting in a complete 'nodes' hashmap.

Direct Distance File Description

The direct distance file contained the node name and the direct distance from the starting node to the destination node, 'Z'.

Class

- Variables

  The first variable is the node map. This consisted of a node and the incidence collection, which is a list of all edges associated with a node.
  Node 'A' is mapped to 'B' - 71 and 'J' - 151.
  Node 'T' is mapped to 'H' - 115 and so on.

```
private static Map<String, HashMap<String, Integer>> nodes = new HashMap<>();

// This is the direct distance map
private HashMap<String, Integer> dd = new HashMap<>();
// This is the nodetracer for Algorithm 1
private static ArrayList<String> nodeTracer = new ArrayList<>();
// This is the nodetracer for Algorithm 2
 private static ArrayList<String> nodeTracer2 = new ArrayList<>();
```

- Constructor

```
public AnotherProject() throws FileNotFoundException { // read in files create nodes and create direct distance map}
```

- Methods
    - Returns a hashmap containing all nodes and associated distances to ‘Z’

```
private static HashMap<String, Integer> readDirectDistance() throws FileNotFoundException
```

- Returns the Weight of the edge between two nodes

```
public Integer weight(String node1, String node2)
```

- Find the shortest edge - helper function

```
public String findShortestEdge(String node)
```

- Find the shortest direct distance from node to ‘Z’

```
public String findShortestDD(String node)
```

- Find the shortest direct distance + weight.. to ‘Z’

```
public String findShortestDD_Plus_Weight(String node)
```

- Clean the node tracer

```
public void cleanNodeTracer()
```

Calling from Main()

- Read all files and build the data structures by instantiating the class
- Prompt User Input for the starting node
- Add starting node to an array list called nodeTracer.
- Algorithm 1:
  Recursively call findShortestDD() until the Target node Z is reached.
- Algorithm 2:
  Recursively call findShortestDDPlusWeight() until the target node Z is reached

Print out results

Testing

Testing the output of this graph began with a trivial straight forward input case that could be verified visually.

The next test case was the dead end node case where the input put the code in a looping state between a dead end node and the parent.
This would be the case in the drawing where you would begin at node T.
All node inputs were tested.

Output

The output is a listing of all visited nodes followed by the node list used for the shortest path.

```
Please enter the name of the Start Node (Case Sensitive):
J
User entered 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
      Cleaning..

Shortest Path: J I L Z
Shortest Path Length: 278


Please enter the name of the Start Node (Case Sensitive):
F
User entered F as the start node

Algorithm 1
Sequence of all nodes: F G H T H L Z
Shortest Path: F G H L Z
Shortest Path Length: 434

Algorithm 2
Sequence of all nodes: F E F G H T H L Z
      Cleaning..

Shortest Path: F G H L Z
Shortest Path Length: 434
```