Spring 2021 - BBM204: Software Practicum Assignment 4

Hacettepe University Computer Engineering Department

Due Date: May 29, 2021 23:59

Introduction

In this assignment, you will practice implementing several data structures and algorithms (quad-trees, ternary search trees, Dijkstra's shortest path algorithm for routing) on a real-world problem. You will build a web application for mapping the real-world map data (tile images and map feature data) which is freely available from the OpenStreetMap project. Your job is to build the back end, i.e. the web server that will drive the API to which requests from the user interface will be sent.

Overview

Your application will allow a user to use their web browser to interface with the web server which will offer similar services to Google Maps (displaying a map, scrolling, zooming, and routing) through an HTML file. The front end code, dataset and starter code (coding template) have been provided to you; hence, you will only work on coding the back end server which does all the mapping and routing work.

You will achieve this functionality by implementing a service that will allow a user to provide a URL via their web browser to your Java program which, in turn, will take this URL and generate the appropriate output to be displayed in the browser. The back end server that you need to build will be responsible for serving a list of images corresponding to a map of the region specified by the user through the front end code. The front end will then display the images to the user through the web browser. The starter code that has been provided to you already handles functionalities of URL parsing and communication with the front end.

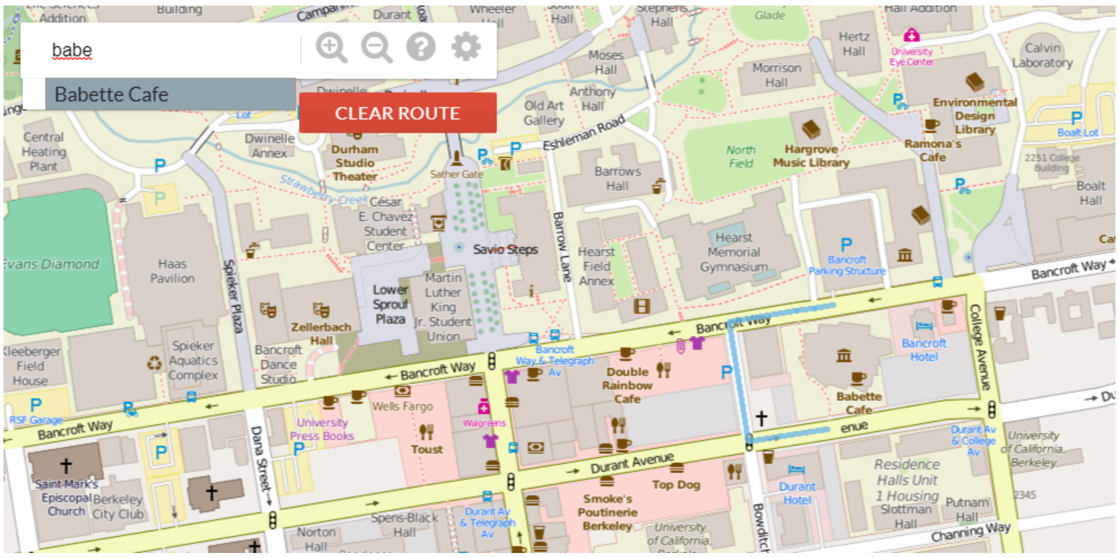
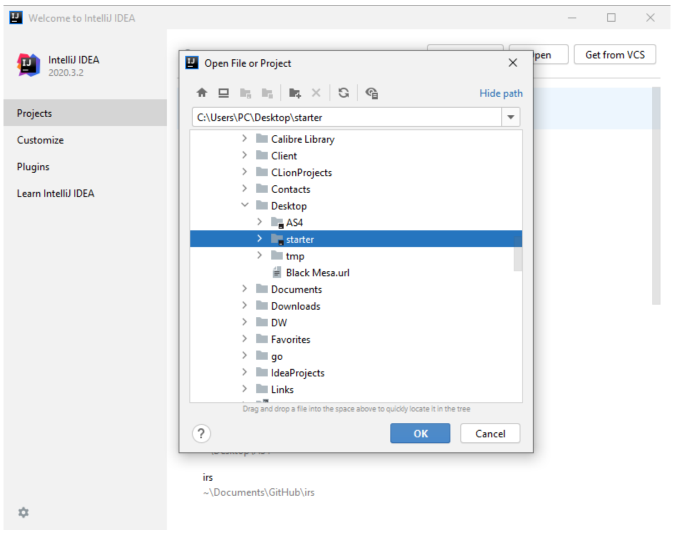


Figure 1: Illustration of the project

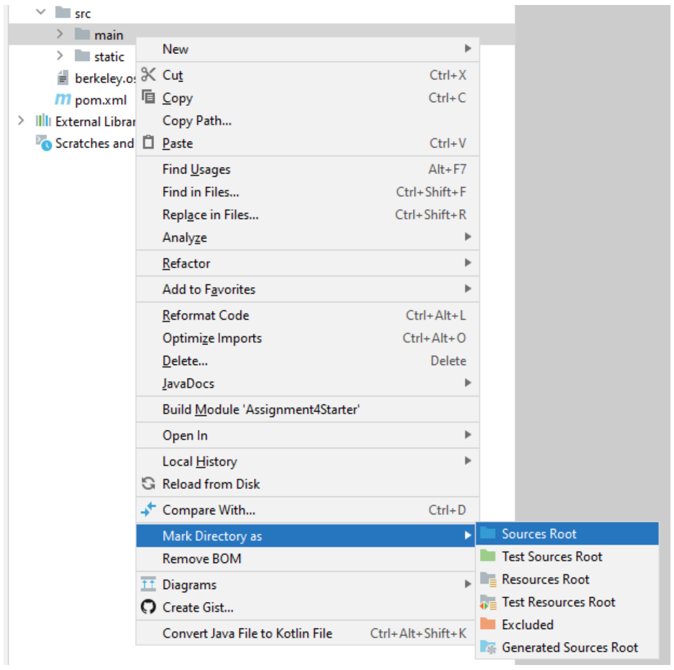
Step 0: Importing the Project

This section illustrates how to import the starter code project using IntellijIDEA which is **HIGHLY** recommended for this project.

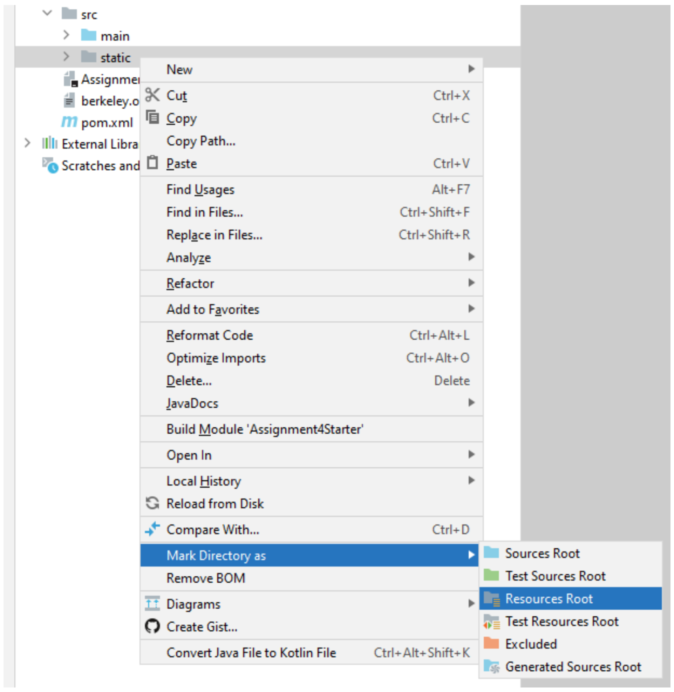
Open the project



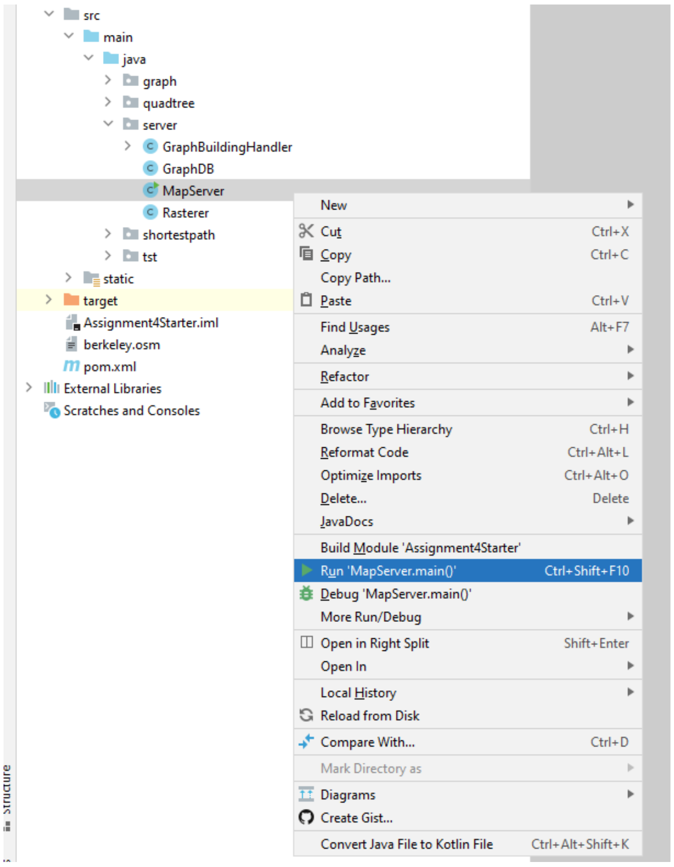
Mark the src directory as the sources root



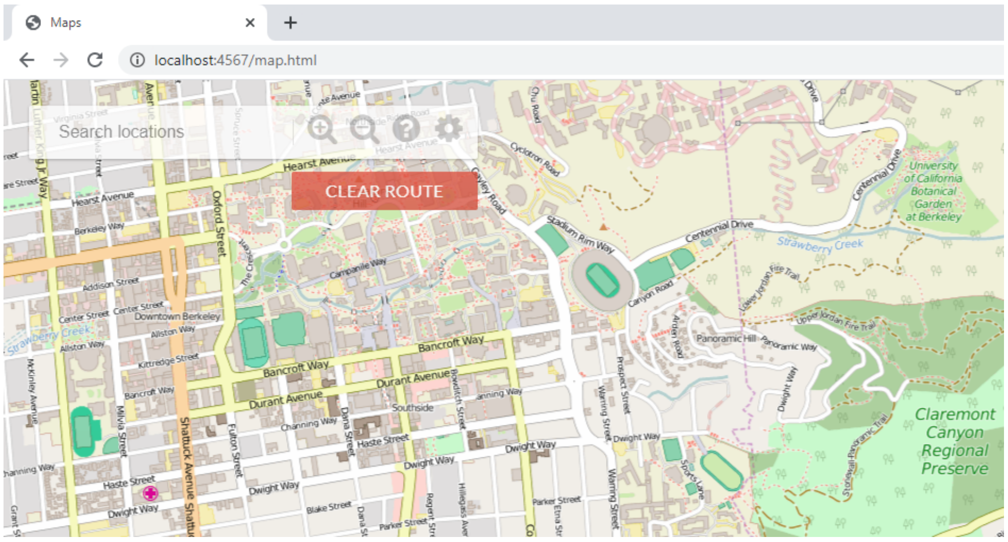
Mark static directory as the resources root and build the project



Run the server



Check the server by opening `localhost:4567` in your browser



Step 1: Serving the Images

The images to be displayed should be served based on a user's interaction with the map. You can think of this as the client side of the application querying for specific boundaries (a "query box") to be displayed based on the user's actions (e.g. zooming in or out, dragging the map). This query box is simply a bounded box defined by two coordinate points: one on the upper left, and another on the lower right corner.

Each image in the directory is a 256X256 PNG file named suitably for a QuadTree construction.

In order to serve the correct images to the client side, you are required to implement a QuadTree. This structure will be used to search for the corresponding images given a query box. You should pay attention to both depth and spatial location of the images during the retrieval process.

To determine which level of images are to be served, you are going to use longitudinal distance per pixel property of both the query box and the images themselves. This value is calculated as follows:

```
lonDPP = (lower right longitude - upper left longitude) / (width of the box in pixels)
```

Building the Tree

You are required to place the images in your implementation of the QuadTree as follows:

- Each node in your QuadTree should include a file name associated with it.
- The tree starts with the root node which should have the value of "root.png".
- In the second level of the tree (children of the root element), there will be 4 nodes for the file names: 1.png, 2.png, 3.png, and 4.png.
- The tree will grow recursively. For instance, the node which contains the file name 1.png will have the following children: 11.png, 12.png, 13.png, and 14.png.
- You can easily tackle this problem by building children of each node recursively.

Along with the file names, you should also store the depth and box properties of each node. The depth can most simply be defined as the following:

root.png $\rightarrow$ 0
1.png $\rightarrow$ 1
11.png $\rightarrow$ 2
...

You should similarly define the box property of each node by using two points: one on the upper left, the other one in the bottom right corner. This value for each node should be determined according to their location in the QuadTree. The box property of the root node is given to you, and you will use it to calculate the box property for other images.

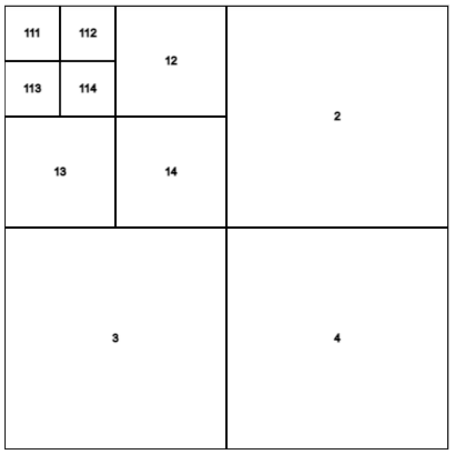


Figure 2: A demonstration of the QuadTree structure

Searching in the Tree

After correctly constructing the tree, you should implement a search mechanism to find and organize necessary files for the given query box.

This mechanism should be implemented in two parts as discussed below.

Searching for Correct Images in the Tree

The first part includes a recursive search in the tree. This process should consider both the lonDPP property (discussed above) and if the images in the tree intersect with the query box. To achieve this, you should retrieve the first depth (zoom level) of the images which intersect with the query box and have a lower lonDPP than the query box.

Organizing the Image List

After you retrieve the list of images using the recursive search mechanism described above, you should order them as a grid. This grid is nothing more than a 2D array of file names. To order the images, you should determine correct row and column for each image (node) in the retrieved list. As a hint, you should consider the latitude values of images to place them in the row, and the file names of the images to place them in a column.

Testing

Upon completion of this step, your application should be able to serve the images to the client side. When you load the page in your browser you should be able to see the map and test the zooming and dragging functionalities.

Step 2: Parsing the XML Data

In this part, you are required to use the supplied SAXParser to handle the results of XML parsing and create a graph for routing. You will also build a ternary search tree for auto-complete functionality.

The XML file you are given to parse is named “berkeley.osm”. In this file, there are *nodes* which represent the vertices of the graph (directed graph as there are both one-way and two-way roads in the data) you are going to create, as well as the *ways* which specify how to connect those vertices.

To achieve these task, you should define some instance variables and add some code to the necessary parts (see coding template for details). SAXParser gives the user two callbacks: *startElement* and *endElement*. The first one tells the user that an XML tag starts, while the second callback tells the user that an XML tag has ended.

In *startElement* callback, you should handle the new node and “*way*” tags. For the case of new nodes, you should add a new vertex to your graph and save other necessary information to be used depending on your implementation (i.e. saving the last saved vertex).

For the case of encountering a “*tag*” tag with the attribute “*name*” while the active state is “*node*”, you should set the name of the current vertex, as well as insert the location name to your ternary search tree. Do not forget to normalize the text by supplying the regular expression given in the coding template.

```
<node id="30365438" lat="37.8229732" lon="-122.3186113" version="1">
            <tag k="ref" v="8A"/>
            <tag k="source" v="http://www.dot.ca.gov/hq/traffops/signtech/calnexus/pdf/80.pdf"/>
            <tag k="exit_to" v="I-880 South;Oakland"/>
            <tag k="highway" v="motorway_junction"/>
    </node>
```

Figure 3: An example node element

For the case of *ways*, there are three scenarios you should handle.

1. The first one is encountering a new “*way*” start tag. In this scenario, you should instantiate a new *Way*, while saving its *id*.
2. The second scenario is encountering a new “*nd*” tag while the active state is ”*way*”. This tag references a node in the current way by its node *id*. When this tag is encountered, you should add the *id* to the *listOfNodes* in that *Way*.
3. The third scenario is encountering a “*tag*” tag while the active state is “*way*”. When this tag is encountered, we can get four types of information using this tag: (i) name of the way, (ii) maximum speed of the way, (iii) if the way is one-directional or not, and most importantly (iv) type of the way (if the way is classified as a highway). At this point, you should check if the parsed highway type is specified to be allowed in the list ALLOWED_HIGHWAY_TYPES. If the type is allowed, you should set a flag as an instance variable to specify that this way should be used to connect vertices in the graph.

```
<way id="6344945" version="1">
        <nd ref="53072584"/>
        <nd ref="93186773"/>
        <nd ref="93186763"/>
        <tag k="name" v="Roble Road"/>
        <tag k="highway" v="residential"/>
        <tag k="maxspeed" v="25 mph"/>
        <tag k="tiger:cfcc" v="A41"/>
        <tag k="tiger:county" v="Alameda, CA"/>
        <tag k="tiger:reviewed" v="no"/>
        <tag k="tiger:zip_left" v="94618"/>
        <tag k="tiger:name_base" v="Roble"/>
        <tag k="tiger:name_type" v="Rd"/>
        <tag k="tiger:zip_right" v="94618"/>
</way>
```

Figure 4: An example way element

In *endElement* callback, you should handle ending of a *way* tag. If the way is marked as a valid one in the third scenario described above, you should connect the vertices specified by the list of references in the *listOfNodes*. In this step, you should be careful if the way is one-directional or not. If the way you were parsing is not one-directional, you should add two edges to your directed graph (in both directions).

Step 3: Implementing the Auto-complete Feature using Ternary Search Trees

In this step, you should implement a ternary search tree to be able to quickly serve the location names that match the requested prefix from the search bar. Be careful and note that you should retrieve the full name of a location when requested, and not the normalized version which is stored in the tree.

Testing

Upon completing this step, your application should be able to serve location suggestions to the user when given a minimum of two-letter prefix. When a location is selected, you should be able to see a marker correctly placed above the selected location on the map (see Fig. 6).

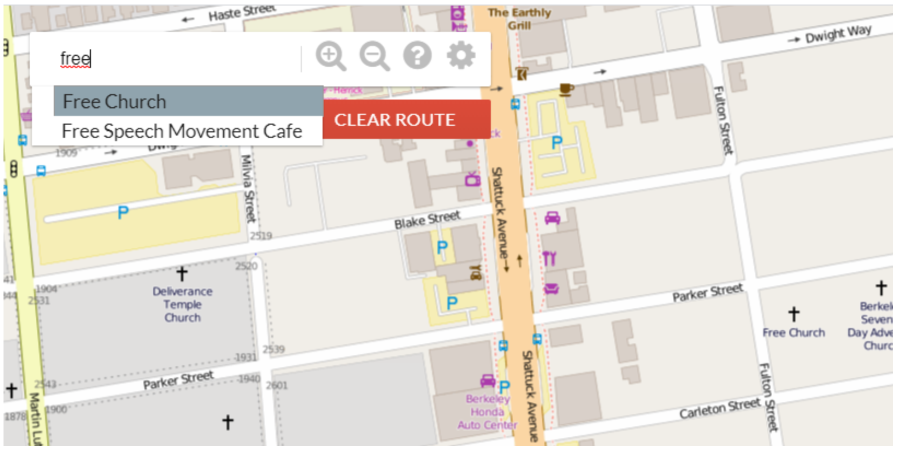


Figure 5: Autocomplete suggestions populating on user input

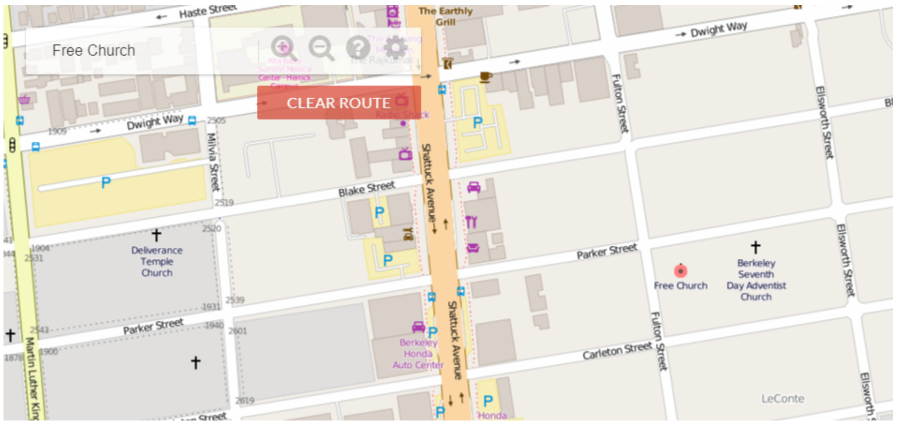


Figure 6: Marker after the selection of a location from the list

Step 4: Dijkstra's Shortest Path Algorithm for Routing

For this step, you should implement a priority queue based Dijkstra's Shortest Path algorithm to calculate a valid route from a source to a destination location. Your code should be able to also handle stop points between the source and destination vertices, when those are supplied by the front end. When a new stop point is selected, your application should run the algorithm again to calculate the shortest path between the stops on the supplied stop list, and update the route accordingly. You should consider the stop points with the increasing order of distance from the start vertex. You can use the pseudo-code given in Figure 8.

Testing

Upon completing this step, your application should be able to display a route for the given source, destination and stop locations. You can test this feature by double-clicking on the map to pick a source location and double-clicking again to pick a destination location. You should be able to add a stop location to the current route by placing the cursor over the location you want to add and then pressing "**s**" on the keyboard.

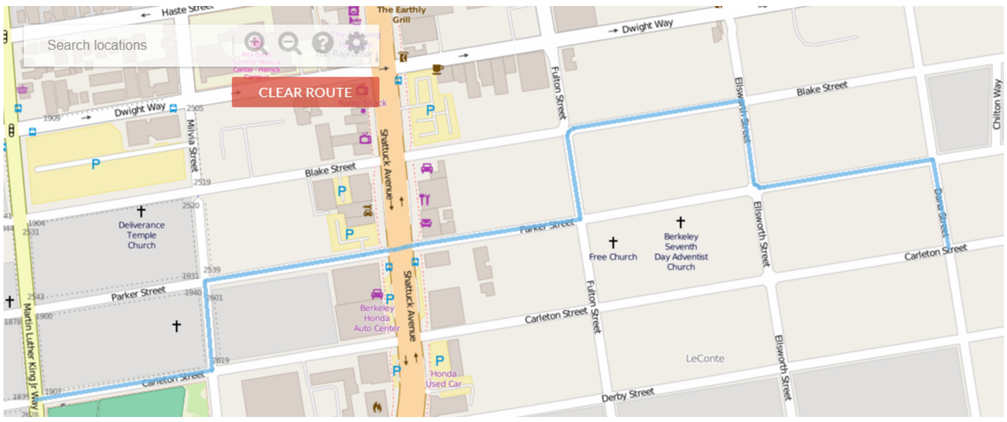


Figure 7: Route with multiple stops

Notes

1. You should submit your code using codepost.io.
2. You should submit your work until 29.05.2021, 23:59.
3. Further instructions about the starter code along with the starter code itself will be given.
4. To acquire full points, your code should pass one or more unit tests for each step.

Grading

Grading will be performed in two parts, comments (10%) and automated tests (90%).

Policy

All work on assignments must be done individually unless stated otherwise. You are encouraged to discuss with your classmates about the given assignments, but these discussions should be carried out in an abstract way. That is, discussions related to a particular solution to a specific problem (either in actual code or in pseudocode) will not be tolerated. In short, turning in work of someone else (from the internet), in whole or in part, as your own will be considered as a violation of academic integrity. Please note that the former condition also holds for the material found on the web as everything on the web has been written by someone else.

Addendum

```
Foreach node set distance[node] = HIGH
SettledNodes = empty
UnSettledNodes = empty

add sourceNode to UnSettledNodes
distance[sourceNode]= 0

while UnSettledNodes is not empty:
    evaluationNode = getNodeWithLowestDistance(UnSettledNodes)
    remove evaluationNode from UnSettledNodes
    add evaluationNode to SettledNodes
    evaluatedNeighbors(evaluationNode)


getNodeWithLowestDistance(UnSettledNodes):
    find the node with the lowest distance in UnSettledNodes and return it


evaluatedNeighbors(evaluationNode):
    Foreach destinationNode which can be reached via an edge from evaluationNode AND which is not in SettledNodes
        edgeDistance = getDistance(edge(evaluationNode, destinationNode))
        newDistance = distance[evaluationNode] + edgeDistance
        if distance[destinationNode]  > newDistance
            distance[destinationNode]  = newDistance
            add destinationNode to UnSettledNodes
```

Figure 8: Pseudocode for the suggested algorithm