AN EVALUATION OF IMPLEMENTING THE TRAVELING SALESMAN PROBLEM UTILIZING THE GOOGLE MAPS API FOR ROUTE OPTIMIZATION

BSc (Hons) Computing with Software Development, 2017
Institute of Technology Tralee

Rafael Lopez

T00185382

Final Year Project

Primary Supervisor: Cathryn Casey

Secondary Reader: Helen Fitzgerald

Abstract

The traveling salesman problem (TSP) is an algorithm problem focused on optimization, and it has become a central part of certain fields including logistics, manufacturing, telecommunications, and robotics. New algorithms have been developed and studied over the last decades in order to find one definite algorithm that can solve every possible instance of the TSP in a timely manner; however, approximation algorithms are still the best option as of now.

The objective of this thesis was to implement three different TSP algorithms (brute force, the nearest neighbor and a genetic algorithm) to address the issue of finding an optimal route for travelers who wish to visit different destinations in Ireland following a distance-effective route. The approach was to develop a Java-based application using the JavaFX technology, which was also connected to different Google Maps API services in order to get all the relevant information to run the algorithms previously mentioned. After the data processing, the application returned a possible optimal path that travelers can follow to visit all the desired destinations only once in a distance-effective way. Additionally a genetic algorithm was implemented in the project using elitism, tournament selection, single-point crossover and swap mutation. The testing on the genetic algorithm was very limited due the Google Maps API usage limit since this project is only using the free version.

Table of Contents

Table of Figures

Table of Code Snippets

Chapter 1: Introduction

Travelling is certainly an activity that more and more people do nowadays, and with greater awareness of CO2 emissions and increasing fuel costs, the need of finding effective routes in terms of distance has become essential. The application let users select all the locations they want to visit and provides them with a distance effective-route. A map will be presented to the user in order to select destinations to be included in a tour, once the user has finished adding locations the application will apply an algorithm based on traveling salesman problem in conjunction with Google maps API services to provide an optimal path to visit all locations only once. As a result, people will travel less distance, reducing the amount of CO2 generated and money spent on fuel.

Chapter 2: The Travelling Salesman Problem

2.1 Introduction

The Travelling Salesman Problem (TSP) is a classic algorithmic problem in the field of computer science focused on combinatorial optimization. According to Wang (Wang, 2014), the problem is described as follows: given a tourist map, a salesman wants to find the optimal Hamiltonian circuit (OHC), i.e., a circuit that visits each city once and exactly once and incurs the least distance, time or cost, etc. According to (Rego, et al., 2011), the TSP can be described as the problem of finding a minimum distance tour of n cities, starting and ending at the same city and visiting each other city exactly once.

2.2 Graph Theory

The TSP can be represented using graph theory. As mentioned in (Rego, et al., 2011), the problem can be defined on a graph G = (V,A), where V = {v1,...,vn} is a set of n vertices (nodes) and A = {(vi,vj)jvi,vj 2 V, i – j} is a set of arcs, together with a nonnegative cost (or distance) matrix C = (cij) associated with A. For example, given the graph shown in Figure 2.1, a possible TSP tour in the graph is 0-1-3-2-0. Therefore, the cost of the tour is 8+10+30+12 = 60.

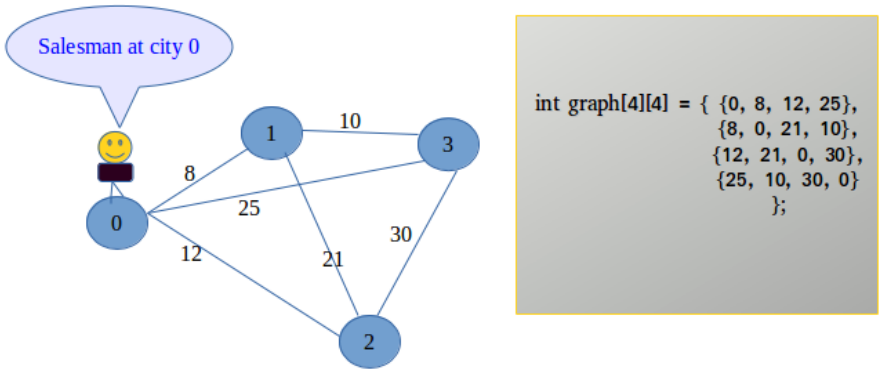


Figure 1 - TSP represented as a graph **(Wikistack, 2016)**

The TSP is classified as an NP-complete problem, and therefore there is not a polynomial-time algorithm able to solve all instances of the problem. Generally, the most efficient solution is not always provided due to the complexity of the problem; instead, different algorithms have been

developed seeking an optimal solution with the lowest computational cost in terms of processing time and resources.

2.3 Classification of the TSP

According to (L. Applegate, et al., 2007), the different instances of the TSP can be classified in two categories. These are:

- Symmetric
- Asymmetric

In a symmetrical TSP, the distance between a pair of cities is the same in each direction, forming an undirected graph.

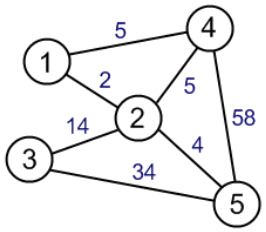


Figure 2 - Symmetrical TSP instance **(Sheard, 2009)**

Figure 2.2 is an example of a symmetrical TSP. Taking node 1 as the departing node, there is a distance of 5 going to node 4, and it is exactly the same distance if node 4 were the initial point and node 1 the end node. As observed in the image, this rule applies to the rest of the nodes, thus this instance falls into the Symmetric category. In (Wang, 2014) , it is described that, given a symmetrical TSP with *n* cities, the number of Hamiltonian Circuits can be calculated by using the following formula:

$$(n - 1)! / 2$$

This is the number of possible paths that can be followed to visit each vertex exactly once.
On the other hand, when the distance is variable between a pair of nodes based on the direction, that instance is considered asymmetrical. In the TSP in particular, traffic accidents, one-way roads, closed streets, etc. are a few examples of reasons as to how symmetrical instances can be transformed into asymmetrical problems.

Based on the formula used to calculate the HCs for a symmetrical TSP, it is deducible that, provided an asymmetrical TSP instance where the distance between each node I different base on the direction, the number of HCs can be calculated using the next formula:

$$(n - 1)!$$

2.4 Type of Algorithms

Since the early days of computers, mathematicians hoped that someone would develop better ways to solve large TSP problems, in other words, algorithms that would allow computers to solve them in a reasonable amount of time. Currently, there are different known approaches to produce an optimal solution, which are explained in the following sections.

2.4.1 Exact solutions

An exhaustive search of all possible paths would guarantee to find the shortest route or cheapest solution. An exhaustive search (also known as "brute force") is the most common type of exact solution for the TSP. However, it is computationally intractable for all instances, only for small sets of locations. For larger problems, optimization techniques are usually needed to intelligently search the solution space and find near-optimal solutions (Sahalot & Shrimali, 2014).

2.4.2 Approximation algorithms

These types of algorithms are designed to find, as their name suggests, approximate solutions. Unlike heuristics, which normally only find reasonably good solutions reasonably fast, this type of algorithms aim for provable solution quality and provable run-time bounds (Lupsa, et al., 2010).

2.4.3 Heuristics

Algorithms that deliver either seemingly or probably good solutions, but which could not be proved optimal. While approximation algorithms are able to produce results that can be

"measured" to see how close they are to the optimal solution, heuristics do not have this property (Glovera, et al., 2001). Examples of heuristics:

- Nearest neighbor
- Greedy
- Insertion heuristics

2.4.4 Metaheuristics

A metaheuristic is a high-level problem-independent algorithmic framework that provides a set of guidelines to develop heuristic optimization algorithms. As mentioned in (Gendreau & Potvin, 2005), a good metaheuristic implementation is likely to provide near-optimal solutions in reasonable computation times. Examples of metaheuristics:

- Tabu search
- Genetic algorithms

2.5 Nearest neighbor

The nearest neighbor algorithm, paraphrasing (Gendreau & Potvin, 2005), is a tour-construction procedure that aims to build an optimal route by taking at each step the node with the cheapest cost or shortest distance from the current node. One of the key points of this algorithm is simplicity given that its methodology is relatively easy in comparison with other algorithms. Figure 2.3 shows the steps performed in the nearest neighbor algorithm.

procedure nearest_neighbor

(1) Select an arbitrary node j, set $l = j$ and $T = \{1, 2, \ldots, n\} \setminus \{j\}$.

(2) As long as $T \neq \emptyset$ do the following.

(2.1) Let $j \in T$ such that $c_{lj} = \min\{c_{li} \mid i \in T\}$.

(2.2) Connect l to j and set $T = T \setminus \{j\}$ and $l = j$.

(3) Connect l to the first node (selected in Step (1)) to form a tour.

end of nearest_neighbor

Figure 3 - Nearest Neighbor pseudocode **(Reinelt, 2003)**

Putting the algorithm in terms of a sales man who needs to visit a set of cities, he would firstly select an aleatory starting point, and then he would visit the nearest city that has not been

visited so far until all cities are visited. Finally, he returns to the starting point to complete the circuit. The complexity of this procedure is O(n2). Now, a possible modification is to consider all vertices as a starting point. The overall algorithm complexity is in turn O(n3), however, the resulting tour is generally better (Laporte, 1992).

2.5 Genetic Algorithm

Based on the natural process of evolution, a genetic algorithm attempts to mimic the concept of generating new populations with more fit individuals, which in the case of the TSP means new solutions that provide shorter tours. This local search algorithm starts by generating an initial population, and then genetic operators are applied to it in order to produce offsprings (new neighborhoods in this context) which are expected to be more fit than their ancestors. At each generation (iteration to produce a new offspring), every new individual represents a possible solution (chromosome). The iteration process continues until the stop criteria is reached, at this point the individual holding the best solution is considered the final result (Pezzellaa, et al., 2007). The different phases of the genetic algorithm are:

1- Initialization: the first population may be generated randomly or by using different methods such as heuristics, and it may be of any size. However, the problem to find a good initial population and the optimal size are complex problems given that these can vary depending on the problem to be evaluated (Diaz-Gomez & Hougen, 2007). Figure 2.4 shows some aspects to take into account when generating an initial population randomly.

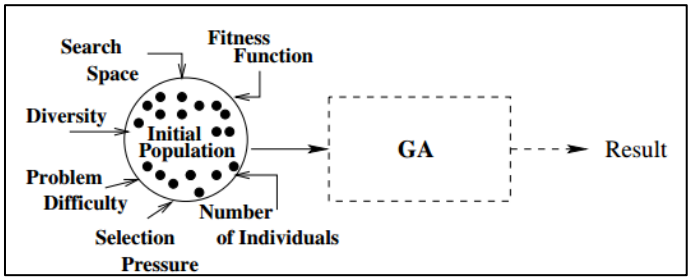


Figure 4 - Aspects to consider for generating an initial population **(Diaz-Gomez & Hougen, 2007)**

2- Evaluation: each individual is evaluated to determine how well it fits the requirements of the problem. In terms of the TSP, the shorter the length of the tour the better.
3- Selection: process of selecting the best chromosomes among the population evaluated. There are different methods (binary tournament selection, roulette wheel selection, etc.) for this, but the idea is the same, make it so that fitter individuals are most likely to be included in the next generation (Alabsi & Naoum, 2012). In other word, a selection mechanism is applied to select individuals from a given population to be inserted into a mating pool. Individuals from this pool are used to produce new offspring; this resulting offspring forms the basis for the next generation. Since the individuals in the mating pool are the ones whose genes are inherited by the following generation, it is desirable that the mating pool be composed of "good" individuals as much as possible (L. Miller & E . Goldberg, 1995).
4- Crossover: aspects (genes) of the individuals selected are combined to generate new members. Methods such as Single Point, Two Points, and Uniform can be applied to do the crossover (Alabsi & Naoum, 2012).
5- Mutation: the intention here is to add a bit of randomness when generating new populations so that the algorithm does not get trapped in a local optimum. Examples of mutation algorithms are Flip Bit, Boundary, etc. (Alabsi & Naoum, 2012)
6- Stopping criterion: the evolution process is repeated until the terminating condition is reached. The number of iterations is commonly the most used stop criteria.

Figure 2.5 shows the pseudo code for a basic genetic algorithm

```
generation = 0;
population[generation] = initializePopulation(populationSize);
evaluatePopulation(population[generation]);
While isTerminationConditionMet() == false do
    parents = selectParents(population[generation]);
   population[generation+1] = crossover(parents);
  population[generation+1] = mutate(population[generation+1]);
   evaluatePopulation(population[generation]);
    generation++;
End loop;
```

Figure 5 - Genetic Algorithm pseudocode (Jacobson & Kanber, 2005)

2.6 Applications

The TSP has been used in several fields where optimization is required. Quoting (L. Applegate, et al., 2007), the TSP has seen applications in the areas of logistics, genetics, manufacturing, telecommunications, neuroscience, and robotics. The majority of the problems that are optimization-related may be good candidates for it.

Chapter 3: Geolocation

3.1 Introduction

Geolocation essentially refers to the process of detecting the real-world geographic position on the Earth of a person or an object through a wireless technology. There are different ways to perform geolocation, but the most common techniques are those that rely on the Internet or satellites.

3.2 Global Positioning System

The U.S. Department of Defense developed the Global Positioning System (GPS) in the late 1960s and early 1970s as a merger of synergistic Navy and Air Force programs for timing and space-based navigation, respectively (McNeff, 2002). It consists of a constellation of 24 satellites, equally allocated in six orbital planes 20,200 kilometers above the Earth. According to (Djuknic & Richton, 2002), satellites transmit the information in two specially coded carrier signals: L1 frequency for civilian use, and L2 for military and government use. GPS receivers process the signals to compute position in 3D using latitude, longitude, and altitude, all this within a radius of 10 meters or better. Figure 3.1 shows how GPS works.

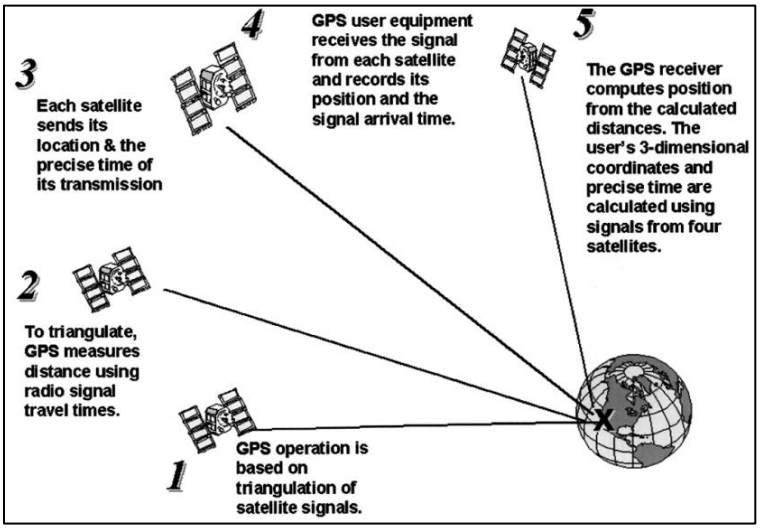


Figure 6 - How GPS works **(McNeff, 2002)**

There are a few things to consider when using GPS. It is important to notice that receivers need a clear view of the skies and signals from at least four satellites. This means GPS not work when

the receiver is inside a building or other radio frequency-shadowed environments. Additionally, its power consumption and cost may not be suitable for all scenarios.

3.3 Network-based technologies

Technologies that rely exclusively on wireless networks often use time of arrival, time difference of arrival, angle of arrival, timing advance, and multipath fingerprinting to provide geolocation. These offer a shorter time-to-first-fix (TTFF) than GPS, a quick deployment and continuous tracking capability for navigation applications. However, network-based technics are far less accurate than GPS, and require expensive investments in base-station equipment (Djuknic & Richton, 2002).

3.4 Google Maps API

As mentioned in (Király & Abonyi, 2015), the Google Maps API is free and publicly available. It provides a fast and reliable web-service for defining user-friendly maps, computing traveling distances and time, and visualizing routes. Furthermore, it is the most popular mapping service nowadays.

3.4.1 How Google Maps and its API work

According to (Svennerberg, 2010), Google Maps is HTML, CSS, and JavaScript working together. The map tiles are images that are loaded in the background with Ajax calls and then inserted into a in the HTML page. As the user navigates the map, the API sends information about the new coordinates and zoom levels of the map in Ajax calls that return new images. That is essentially how Google Maps works. In terms of the API, it consists of JavaScript files that contain classes, methods and properties that users can call to tell the map how to behave and extract information from it.

Google Maps also offers a collection of web services (HTTP interfaces) to Google services that provide geographic data for maps applications (Google, 2017). The different services available to the public are listed below:

- Directions API: a service used to calculate directions between locations. Modes of transportation such as driving, walking, or cycling can be specified in the search (Google, 2017).
- Distance Matrix API: a service to calculate time and travel distance for a given matrix containing origins and destinations, based on the recommended route between start and end points (Google, 2017).
- Elevation API: the Elevation API offers elevation data for locations on the surface of the earth, including depth locations on the ocean floor (Google, 2017).
- Geocoding API: the Geocoding API service provides geocoding and reverse geocoding of addresses functionalities. The process of transforming addresses (e.g. street addresses) into geographic coordinates (e.g. latitude and longitude), is known as geocoding and it can be used to place markers on a map, or position the map. The process of transforming geographic coordinates into a human-readable address is known as reverse geocoding (Google, 2017).
- Geolocation API: the Geolocation API provides a location and accuracy radius based on information about cell towers and WiFi nodes that the mobile client can detect (Google, 2017).
- Roads API: the Roads API identifies the roads a vehicle was traveling along and provides extra metadata about such roads (e.g. speed limits) (Google, 2017).
- Time Zone API: the Time Zone API provides time offset data for locations on the surface of the earth. Information such as name of the time zone, the time offset from UTC, and the daylight savings offset is can be retrieved from this service (Google, 2017).
- Places API Web Service: this service provides up-to-date information about locations: points of interest and businesses (Google, 2017).

Google Maps API web services are free for a wide set of use cases with established usage limits. Paid plans including annual contracts are available for extended usage should it be required (Google, 2017).

3.4.2 Coordinates

It is widely known that coordinates are used to express locations in the world. There are several different coordinate systems. The one used in Google Maps is the Word Geodetic System 84 (WGS 84), which is the same system the Global Positioning System (GPS) uses. In this particular case, the coordinates are expressed using latitude and longitude. This is how locations are described and, therefore, expressed in Google Maps.

Chapter 4: Methodology & Design

4.1 Research Methodology

The research methodology for this thesis involved investigating the traveling salesman problem and the different types of algorithms used to generate optimal solutions, focusing mainly on three particular instances: brute force, the nearest neighbor and genetic algorithms. After analyzing these three different algorithms, it was concluded that it should be feasible to use them in a route optimization solution to process a set of destinations and provide an optimal tour to visit them all.

Geolocation and the Google Maps API in particular, is the other area of research. After considering the power of Google Maps API, it was considered that it could be integrated to a GUI application to select different locations, and get their coordinates afterwards. This information can be then used as input for the three algorithms mentioned earlier.

4.2 Research Question

An evaluation of implementing the Traveling Salesman problem utilizing the Google Maps API for route optimization.

4.3 Proposed Solution

The proposed solution is to develop a Java desktop application that will present a map of Ireland to the users, who will then be able to add destinations to be provided with a time-effective route. Screenshots are included in this section to describe how users will interact with the application and what it would look like.

This is the screen displayed when the application is launched. It includes:

- Name of the app
- Map of Ireland
- List to hold selections

- Four buttons

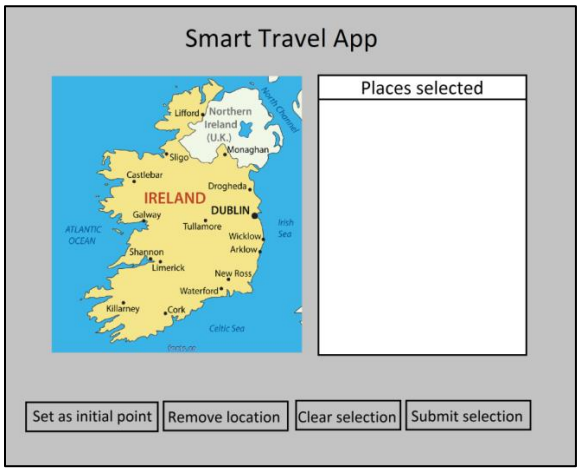


Figure 7 - Prototype: Screen presented to users

To add a destination to the list, users can simply click on the map and it will be added to the list on the right. This is how users will add all their destinations to the tour.

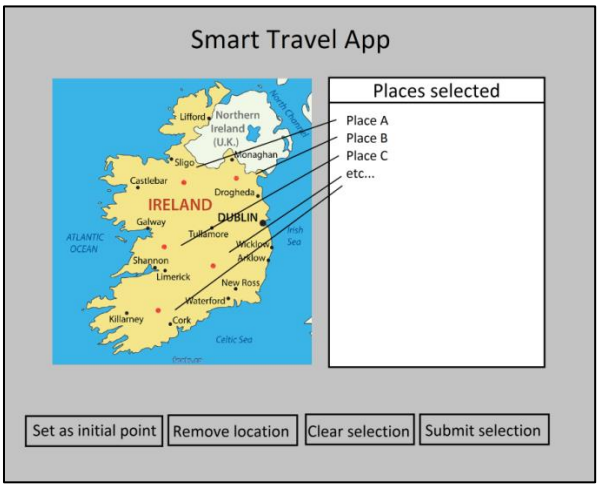


Figure 8 - Prototype: Selecting a destination

To set a destination as the starting point, users will need to select the location in the list, after users can click the Set as initial point button to make the selected location the starting point of the tour and the record will be displayed in red.

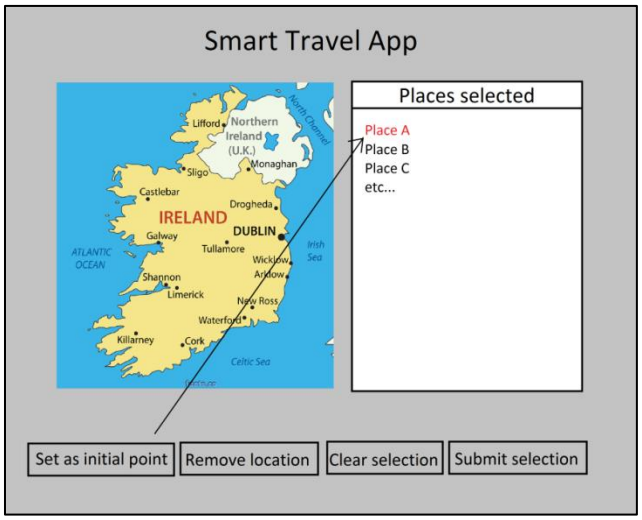


Figure 9 - Prototype: Selecting an initial point

To remove a destination from the list, the user had to select it and then click the "Remove" location button. After this, the location will be removed from the list as shown in the screenshot.

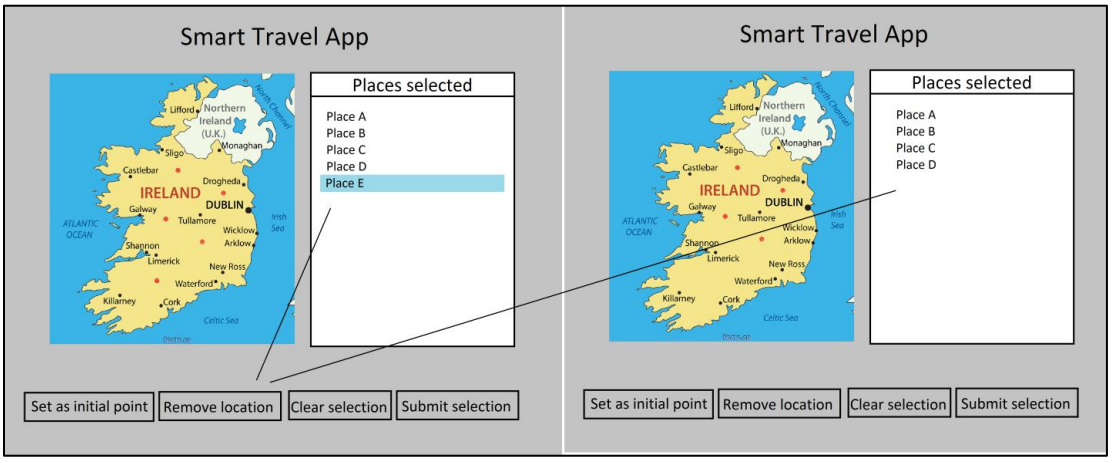


Figure 10 - Prototype: Removing one location from the list

To clear the list of locations added to the tour, the user only needs to click the Clear selection button. All locations will be removed as shown in the screenshot.

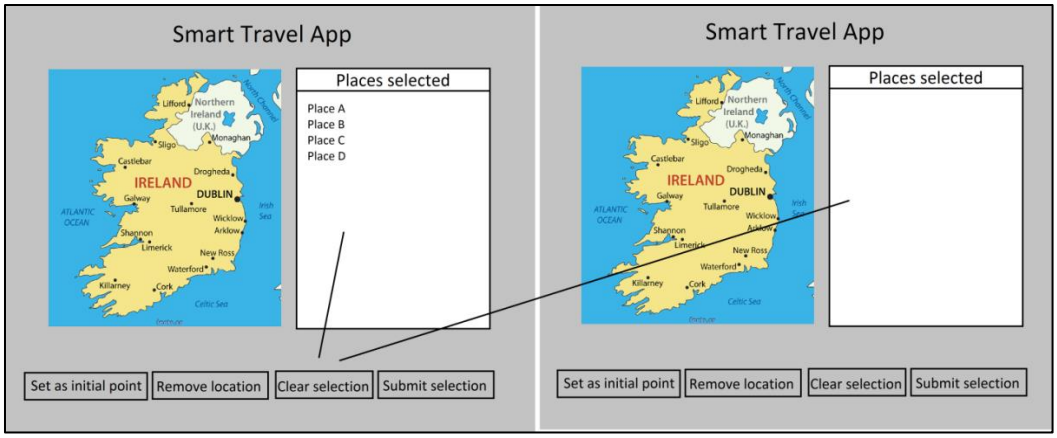


Figure 11 - Prototype: Clearing entire selection

Finally, users will be able to submit their current selection to be processed. The result will be displayed in a pop up window and it will include all locations selected in the order provided after applying one of the algorithms, as well as the total distance of kilometers to travel.

An Ok button is added on the window so that the user can close it and proceed with a different selection.

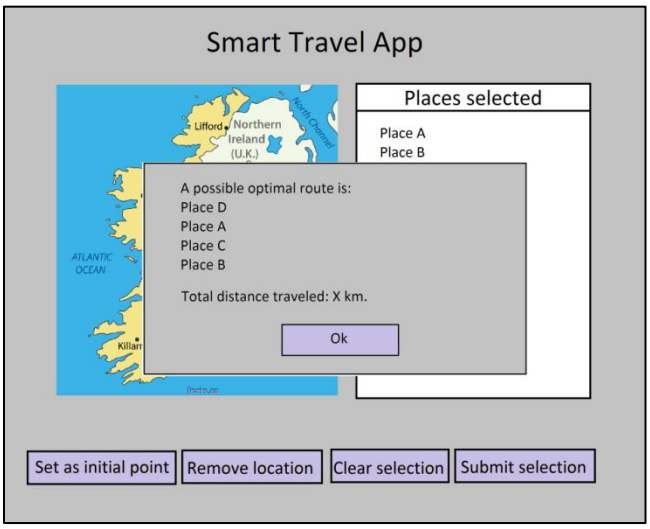


Figure 12 - Prototype: Displaying result

The minimum number of destinations required to process a request is 3. Otherwise, an error message is displayed to let the user know more destinations are needed.

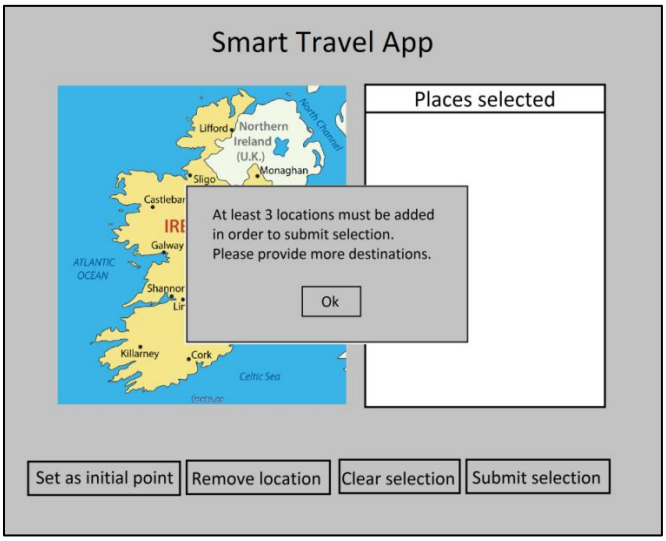


Figure 13 - Prototype: Not enough destinations to process request

4.4 Project Scope

The objective of this project consists of developing a GUI application that will display a map of Ireland where users will be able to add destinations to finally be given a time-effective route. The routes will be generated by applying one of the following algorithms: brute force, nearest neighbor or a genetic algorithm. However, this will be completely hidden from users, as the algorithm selection will be made internally based on the number of locations selected so that the response can also be provided in a timely manner. The functionalities users will be able to utilize are:

- Add a location to visit by selecting it on the map
- Select a location as the starting point for the tour
- Remove a selected location from the list
- Clear entire selection
- Submit selection to be processed

4.5 User Description

The audience for this application is every traveler that desires to visit many places in Ireland without passing by the same destination twice following a time-effective path. They will be beneficiated by saving time, money and the effort needed to plan a travel route.

4.6 List of Application Features

All features are specified following the MoSCoW method, and classified as MUST HAVE, SHOULD HAVE, COULD HAVE and WON'T HAVE.

4.6.1 MUST HAVE

ID	MUST HAVE
001	Implement Brute Force algorithm
002	Implement Nearest neighbor algorithm
003	Implement a Genetic algorithm
004	Display map of Ireland
005	Allow users to add destinations
006	Submit user selection and apply algorithm
007	Display results generated after applying an algorithm
008	Get coordinates of selected locations
009	Make HTTP requests through Google Maps API

4.6.2 SHOULD HAVE

ID	Feature
010	Use one of the 3 algorithms to calculate an optimal route
011	Clear existing selection
012	Not crash at any moment
013	Display total number of kilometers to travel
014	Delete one selection at a time
015	Read JSON result from Google Maps API calls
016	Display error message when user submits less than 3 locations
017	Allow users to set an initial point for tour

4.6.3 COULD HAVE

ID	Feature
018	Save results for future similar requests
019	Suggest popular places

4.6.4 WON'T HAVE

ID	Feature
020	Include maps of other countries
021	Database integration

4.7 Prototype

The prototype consists of a basic user interface that allows to select a file containing a symmetric matrix, which represents the nodes (locations) for a TSP instance. It also displays two text fields, one to specify the number of nodes and another one to select the starting point for the tour.

Task Number	Details	Status
1	Nearest neighbor algorithm implementation	Complete
2	Generate/get test data	Complete
3	Create simple user interface to process test data	Complete

Test Data

Test data is stored in a simple ".txt" file, which contains a symmetric matrix representing the different locations to visit for a particular instance. The number of columns/rows is equals to the number of vertices (destinations), and the edges (distances) are specified for each pair of vertices.

For an instance with four vertices (A, B, C and D) similar to the one below, the test file would look like this:

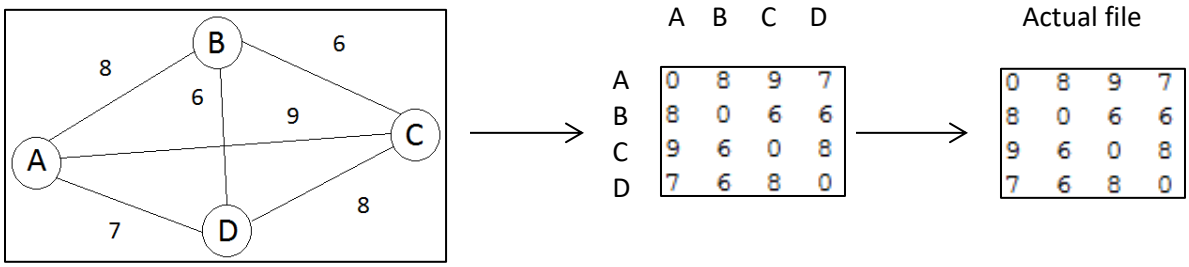


Figure 14 - Prototype file sample

User interface

The user interface has a File menu, where users need to select what test data file to process. After doing so, the number of nodes and starting point must be specified. Note the number of vertices must match the number of vertices included in the sample date.

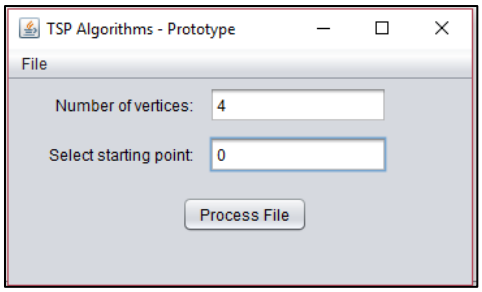


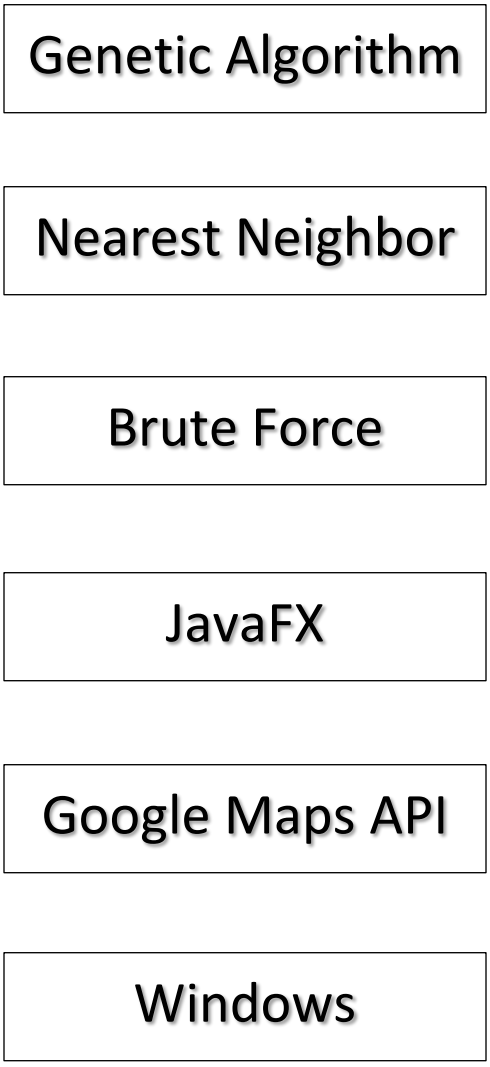
Figure 15 - Prototype Java GUI

Once the test file has been selected and the required information provided, the user may click Process File. Continuing with the example of the four vertices (A, B, C and D) and using vertex A as the starting point, the output should be similar to:

```
*****Neareast Neighbor Algorithm******
Total distance: 19
Path:
0
3
1
2
```

Figure 16 - Prototype output sample

4.8 System Architecture Diagram



Chapter 5: Implementation

The implementation of this project will be carried out using an Agile methodology. Sprints with defined tasks will be used to formalize this process. Each sprint will have a set of tasks that should be completed during the current sprint. Note that sprints may be modified as required.

5.1 Sprints

Each sprint will be 2 weeks long, and will have established goals to be fulfilled after the sprint has ended. However, sprints may be modified accordingly under certain circumstances such as unfinished work, blocking tasks, etc.

5.2 Sprint schedule

There will be seven sprints in total. The sprint schedule is displayed below:

Sprint Number	Start Date	Finish Date
1	23/01/2017	03/02/2017
2	06/02/2017	17/02/2017
3	20/02/2017	03/03/2017
4	06/03/2017	17/03/2017
5	20/03/2017	31/03/2017
6	03/04/2017 24/04/2017	07/04/2017 28/04/2017

5.2.1 Sprint One

Integrate the Google Maps JavaScript API service and Java to be able to include an interactive map within a Java-based application.

5.2.1.1 Google Maps JavaScript API and Java integration

Task	**Status**
Get Google Maps JavaScript API key	Complete
Create Java application that will contain the map	Complete
Load and display map within the Java application	Complete
Set up map to display Ireland when loaded	Complete
Test application to make sure users can zoom-in/out and drag the map around	Complete

The first sprint required to combine Java and the Google Maps JavaScript API service in order to display a map that will then allow users to select locations. Getting a key for the Google service was relatively easy; the developer simply needs to request one through the Google Maps API website.

The next step was to find a way to combine Java and Google Maps to produce an application that would act as a container for the map. After some research, it was determined that JavaFX was the simplest solution for this scenario mostly because it comes with WebView, which according to Oracle's description, is a web component that uses WebKitHTML technology to make it possible to embed web pages within a JavaFX application. Additionally, JavaScript running in WebView can call Java APIs, and Java APIs can call JavaScript running in WebView (Oracle, 2013). With all this in mind, JavaFX is a perfect fit for this case. Creating a JavaFX project is similar to creating a regular Java application in NetBeans, the only difference is that

JavaFX must be selected instead of Java when creating the project as displayed in the image below.

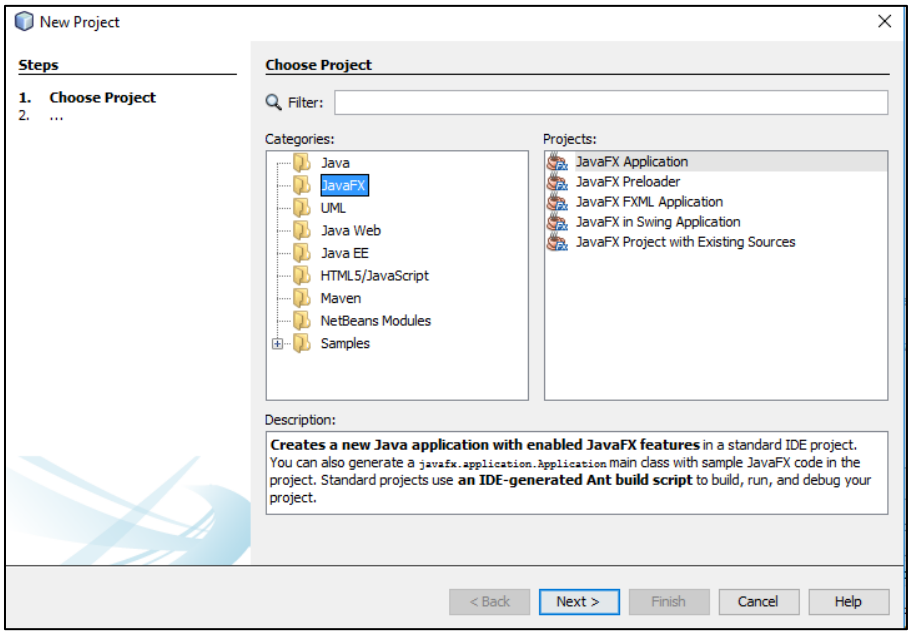


Figure 17 - Creating JavaFX project in NetBeans

After creating the JavaFX project, the default class added by NetBeans will extend "Application", and will have a start() method, which must be overwritten as it is the method that gets called when the application is launched. This is the method where the properties (window size, title, etc.) are usually set up for the application.

```
public class JavaFXTest extends Application {

    private Scene scene;
    private MyBrowser myBrowser;

    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("Google Maps API Demo");
        myBrowser = new MyBrowser();
        VBox mainbox = new VBox();
        mainbox.getChildren().add(myBrowser);
        scene = new Scene(mainbox, 1200, 680);
        primaryStage.setScene(scene);
        primaryStage.show();
    }
```

Code Snippet 1 shows the start() method in main class

Furthermore, the main method will call the inherited launch() method, which is used to load a standalone application. By calling it, the start() method previously discussed is called in consequence.

```
public static void main(String[] args) {

    launch(args);
}
```

Code Snippet 2 shows main method

Next, loading and displaying a map within the JavaFX application is achieved through the load() function defined in the WebEngine class. First, an HTML page called Map.html was created where the Google Maps JavaScript API service was used to add an interactive map. The code for this HTML page is similar to this:

```
<!DOCTYPE html>
<html>
  <head>
    <style>
       #map {
        height: 400px;
        width: 100%;
       }
    </style>
  </head>
  <body>
    <h3>My Google Maps Demo</h3>
    <div id="map"></div>
    <script>
      function initMap() {
        var uluru = {lat: -25.363, lng: 131.044};
        var map = new google.maps.Map(document.getElementById('map'), {
          zoom: 4,
          center: uluru
        });
        var marker = new google.maps.Marker({
          position: uluru,
          map: map
        });
      }
    </script>
    <script async defer
    src="https://maps.googleapis.com/maps/api/js?key=YOUR_API_KEY&callback=initMap">
    </script>
  </body>
</html>
```

Code Snippet 3 shows HTML to display map (Google, 2017)

There are two script tags defined, one is to define the initMap() function, which initializes and adds the map when the web page is loaded (here is where the latitude and longitude values are specified so that the map shows Ireland when loaded). The second script loads the Google Maps API from the specified URL. In addition, the callback parameter executes the initMap() function after the API is completely loaded. Finally, the key parameter contains the API key obtained after completing the first task of this sprint.

After creating the HTML page that will contain the map, a new class called MyBrowser was defined, which extends Region, the base class for all JavaFX Node-based UI Controls and all layout containers (Oracle, 2015). Then a URL object was defined to specify the URL for the HTML webpage. This object is passed to the load() function previously mentioned so that the map can be displayed.

```
public class MyBrowser extends Region {
    WebView webView = new WebView();
    WebEngine webEngine = webView.getEngine();

    public MyBrowser() {
        webView.setMinSize(700, 340);
        final URL urlGoogleMaps = getClass().getResource("Map.html");
        webEngine.load(urlGoogleMaps.toExternalForm());

        getChildren().add(webView);
```

Code Snippet 4 shows code to embed map into the Java application

Once the map was added to MyBrowser, an instance of this class was created in the main class and attached to the main scene so that when the application was loaded it would display the HTML page showing the map.

```
    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("Google Maps API Demo");
        myBrowser = new MyBrowser();
        VBox mainbox = new VBox();
        mainbox.getChildren().add(myBrowser);
        scene = new Scene(mainbox, 1200, 680);
        primaryStage.setScene(scene);
        primaryStage.show();
    }
```

Code Snippet 5 shows code to build and display UI

The result of running this code is a window with an interactive map that displays Ireland.

Figure 17 - GUI after first sprint

The zoom and dragging functionalities were tested. They worked as expected. It would be ideal to limit the dragging functionality so that users can only drag the map within the region of Ireland in order to prevent users from selecting locations in other countries.

5.2.2 Sprint Two

Ability to add locations to a tour, get coordinates and display current selection

5.2.2.1 Establish communication between both sides HTML and JavaFX to be able to perform Java method calls.

Task	**Status**
Invoke Java methods from the HTML/JavaScript end	Complete
Add a button to add a location to the tour	Complete
Obtain coordinates of a selected place on the map	Complete
Store places' information on the Java side	Complete

Add a button to print all current selected location	Complete

The most challenging part of the second sprint was to be able to establish a way to communicate between HTML/JavaScript and Java. This part was accomplished by using the WebEngine class and its method called executeScript(), along with the JSObject class and its method setMember(). Note that it is necessary to get a WebEngine instance by calling the getEngine() method in the WebView class. Additionally, the parameter passed into the executeScript() method represents the context in which the script will be executed, for this particular scenario "window" represents the HTML page displaying the map. Finally, the setMember() method requires two parameters, a name for the JavaScript property to be accessed and the actual value. The code is displayed in the following image.

```
public class MyBrowser extends Region {
    WebView webView = new WebView();
    WebEngine webEngine = webView.getEngine();

    public MyBrowser() {
        webView.setMinSize(700, 340);
        final URL urlGoogleMaps = getClass().getResource("Map.html");
        webEngine.load(urlGoogleMaps.toExternalForm());

        getChildren().add(webView);

        //setMember() should be called after the page has been loaded. The reason is, JavaScript world is recreated
        //each time a new page is loaded. The newly created window object won't have any custom members installed. A
        //fine place to call setMember() is from a listener attached to WebEngine.getLoadWorker().stateProperty().
        JSObject jsobj = (JSObject) webEngine.executeScript("window");
        jsobj.setMember("java", new JavaScriptHandler());
    }

}
```

Code Snippet 6 shows code to create a bridge between the HTML/Javascript and Java sides

Note that a JavaScriptHandler reference is sent in as the JavaScript value to be accessed from the HTML/JavaScript side. This is a customized class will contain all the methods required to do the information processing from the map such as adding locations, deleting all locations selected, etc. At this stage, it only contains two methods, one to add locations and one to print all selected locations.

```
public class JavaScriptHandler
{
    LocationsContainer locations;

    public JavaScriptHandler()
    {
        locations = new LocationsContainer();
    }

    public void addLocation(String longitude, String latitude)
    {
        locations.add(new Location(longitude, latitude));
        System.out.println("Location added");
    }

    public void printLocations()
    {
        Iterator e = locations.iterator();
        while (e.hasNext())
        {
            Location location = (Location) e.next();
            System.out.println(location);
        }
    }
}
```

Code Snippet 7 shows JavaScriptHandler class after second sprint

Once it was possible to make Java calls from the HTML/JavaScript side, the next step was to add two buttons, one to add locations and another one to print them all. Both are HTML buttons and call JavaScript functions that then call the Java methods.

```
    function addLocation()
    {
        java.addLocation(longitude, latitude);
    }

    function printLocations()
    {
        java.printLocations();
    }
</script>
<input id="addLocation" type="button" value="Add Location" onclick="addLocation();" />
<input id="printLocations" type="button" value="Print Locations" onclick="printLocations();" />
<script async defer
  src="https://maps.googleapis.com/maps/api/js?key=AIzaSyA-OR18X50QuLmTyhP4izLzCyWcza_eDGM&librari
</script>
```

Code Snippet 8 shows Javascript code to call function in JavaScriptHandler class

One of the important goals of this sprint was to be able to click on the map and get the coordinates of the selected location. This was accomplished by placing a listener on the map to catch events generated by clicking on it, and then storing the longitude and latitude into two variables that are used as parameters to the addLocation() method declared in JavaScriptHandler. From Google documentation: a UI 'click' event typically passes a MouseEvent containing a latLng property denoting the clicked location on the map (Google, 2017).

```
    //Listener
    map.addListener('click', function(e)
    {
        latitude = e.latLng.lat();
        longitude = e.latLng.lng();
        placeMarkerAndPanTo(e.latLng, map);
    });
}
```

Code Snippet 9 shows listener placed on the map to obtain coordinates

After finishing sprint two, the application looked like this:

Figure 18 - GUI after second sprint

5.2.3 Sprint Three

Integrate Nearest Neighbor algorithm into the application to process current selection and produce an optimal route.

5.2.3.1 Nearest Neighbor algorithm integration

Task	**Status**
Add pre-developed nearest neighbor algorithm to main project	Complete
Add a button to process current selection	Complete
Process selection by calling the Nearest neighbor algorithm	Complete

Display results	Complete

A Nearest Neighbor implementation was developed for the prototype of this project; therefore, this existing code was taken from the prototype and added to the main project. A new package called “algorithms” was added under the “utils” package, and its purpose is to hold different algorithm implementations. In this particular case, the three different algorithms (brute force, nearest neighbor and genetic) will be located in this package.

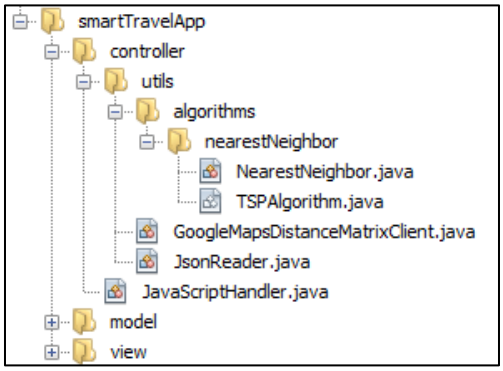


Figure 19 - Algorithms package

It is also important to mention that, every algorithm to be added to this package must extend TSPAlgorithm. This is an abstract class designed following the Template Method design pattern, whose intent is

“Define the skeleton of an algorithm in an operation, deferring some steps to client subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure (Sierra, et al., 2004)*.”*

The processInstance() method represents the skeleton of the algorithm, and it takes in two parameters, the locations to be processed and a starting node. In the code below, it can be observed that the first step is to generate a symmetric matrix containing all the different distances between each pair of nodes. To calculate the distance between each pair of nodes the Google Maps Distance Matrix API was used, which is a service that provides travel distance and

time for a matrix of origins and destinations (Google, 2017). A sample URL request for this service looks like this:

https://maps.googleapis.com/maps/api/distancematrix/json?origins=52.65639394198803,-8.63525390625&destinations=52.649729197309426,-7.239990234375%7c53.35710874569601,-6.273193359375&mode=walking&language=en-EN&key=AIzaSyA-ORl8X50QuLmTyhP4izLzCyWcza_eDGM

The JSON result is similar to:


```
{
   "destination_addresses" : [
      "Canal Walk, Lacken, Co. Kilkenny, Ireland",
      "Phibsborough Rd, Arran Quay, Dublin, Ireland"
   ],
   "origin_addresses" : [ "Saint Alphonsus Villas, S Circular Rd, Limerick, Ireland" ],
   "rows" : [
      {
         "elements" : [
            {
               "distance" : {
                  "text" : "108 km",
                  "value" : 108146
               },
               "duration" : {
                  "text" : "22 hours 2 mins",
                  "value" : 79346
               },
               "status" : "OK"
            },
            {
               "distance" : {
                  "text" : "192 km",
                  "value" : 191672
               },
               "duration" : {
                  "text" : "1 day 15 hours",
                  "value" : 141108
               },
               "status" : "OK"
            }
         ]
      }
   ],
   "status" : "OK"
}
```


Figure 20 - Distance Matrix API output sample

Once the distance array is generated, it is sent in to the processTour() method as a parameter along with the starting node, and finally the formatMessage() method produces a String to be displayed as the final result.

```
public abstract class TSPAlgorithm
{

    /**
     * A method that takes in a list of locations as part of a tour
     * and returns an optimal distance-effective path.
     * @param locations Locations to be processed
     * @param startingNode Starting node for the tour
     * @return
     */
    public final String processInstance(LocationsContainer locations, int startingNode)
    {
        long[][] distances = getSymetricMatrix(locations);
        Integer[] solution = processTour(distances, startingNode);
        return formatMessage(solution, locations);
    }
```

Code Snippet 10 shows TSPAlgorithm abstract class

Note that processTour() is an abstract method to be implemented by every subclass extending TSPAlgorithm.

```
abstract Integer[] processTour(long[][] data, int start);
```

Code Snippet 11 shows abstract method processTour() defined in TSPAlgorithm class

Now, to have a better understanding of how the Nearest Neighbor was coded, it is imperative to remember the steps taken in such an algorithm. Figure 5.1 illustrates the process involved.

procedure nearest_neighbor

(1) Select an arbitrary node j, set $l = j$ and $T = \{1, 2, \ldots, n\} \setminus \{j\}$.

(2) As long as $T \neq \emptyset$ do the following.

(2.1) Let $j \in T$ such that $c_{lj} = \min\{c_{li} \mid i \in T\}$.

(2.2) Connect l to j and set $T = T \setminus \{j\}$ and $l = j$.

(3) Connect l to the first node (selected in Step (1)) to form a tour.

end of nearest_neighbor

Figure 21 - Nearest Neighbor pseudocode (Reinelt, 2003)

The first step is to select a starting node, at this point in the project this is manually set up to be always the first node selected (users will be given the ability to select a starting point later on). Using this starting point, the next step is to find the closest node (or nearest neighbor), which is carried out by the findNearestNeighbor() method. Once that is done, this node is added to the solution and the code keeps going on until every node is visited and there are no nodes left out.

```
public class NearestNeighbor extends TSPAlgorithm
{
    @Override
    public Integer[] processTour(long[][] data, int start)
    {
        Integer[] solution = new Integer[data.length];
        long distance = 0;
        int currentNode = 0;

        for (int i = 1; i < data.length; i++)
        {
            if (i == 1)
            {
                solution[0] = start;
                currentNode = start;
            }

            HashMap results = findNearestNeighbor(data, currentNode, solution);
            distance += (Long)results.get("distance");
            currentNode = (Integer)results.get("index");
            solution[i] = currentNode;
        }
        System.out.println("*****Neareast Neighbor Algorithm******");
        System.out.println("Total distance: " + distance);
        return solution;
    }
```

Code Snippet 12 shows code for NearestNeighbor class

It is essential to point out that, since it is assumed users do not want to go back to the starting point, the last step of the algorithm is not taken into account (connect the last node to the initial node).

The last part consisted in adding a button to invoke the nearest neighbor algorithm and use it to process the current selection on the map. The Java implementation can be seen below:

```
public void processSelection()
{
    String message;
    if (!locations.isEmpty())
    {
        TSPAlgorithm algorithm = new NearestNeighbor();
        message = algorithm.processInstance(locations, 0);
        JOptionPane.showMessageDialog(null, message, "Result", 1);
    }
    else
    {
        JOptionPane.showMessageDialog(null, "No locations selected!", "Error", 0);
    }
}
```

Code Snippet 13 shows code to process tour

The HTML implementation is displayed in the image below:

```
    function processSelection()
    {
        java.processSelection();
    }
</script>
<input id="addLocation" type="button" value="Add Location" onclick="addLocation();" />
<input id="printLocations" type="button" value="Print Locations" onclick="printLocations();" />
<input id="clearSelection" type="button" value="Clear Selection" onclick="clearSelection();" />
<input id="processSelection" type="button" value="Process Selection" onclick="processSelection();" />
```

Code Snippet 14 shows HTML code to add Process Selection button

5.2.4 Sprint Four

Develop Genetic Algorithm

5.2.4.1 Nearest Neighbor algorithm integration

Task	**Status**
Develop genetic algorithm	Incomplete

Subtask: Implement Tournament Selection as the selection method	Complete
Subtask: Implement Single Point crossover as the crossover method	Complete
Subtask: Implement Swap mutation as the mutation method	Incomplete

Note: the development of the genetic algorithm for this project was dived into three subtasks, each of one corresponding to the different steps carried out in the GA (selection, crossover, and mutation). Figure 5.3 shows the pseudo code for a basic genetic algorithm

```
generation = 0;
population[generation] = initializePopulation(populationSize);
evaluatePopulation(population[generation]);
While isTerminationConditionMet() == false do
    parents = selectParents(population[generation]);
   population[generation+1] = crossover(parents);
  population[generation+1] = mutate(population[generation+1]);
   evaluatePopulation(population[generation]);
    generation++;
End loop;
```

Figure 22 - Genetic Algorithm pseudocode (Jacobson & Kanber, 2005)

According to the algorithm shown above, the first step is to generate an initial population, which will then be evaluated until the termination condition is reached. Therefore, a new class called GeneticAlgorithm (such class extends the TSPAlgorithm abstract class) was created in a new package "geneticAlgorithm" under model.algorithms, and a method named "generateInitialPopulation()" was coded within this class. The code for this method is displayed below.

```
private Population generateInitialPopulation(LocationsContainer locations)
{
    int populationSize = locations.size() * 2;

    Population population = new Population();
    population.initializePopulation(locations, populationSize);
    return population;
}
```

Code Snippet 15 shows code to generate initial population

The population size was set to be twice the number of locations selected by the user considering that it may be of any size and generated randomly or by using different methods such as heuristics (page 10). The actual population is being created by the method initializePopulation() defined in the Population class, which is using a private method to generate individuals for the population, tours in this case.

```
lic class Population

ArrayList<Tour> population;

public Population()
{
    population = new ArrayList<>();
}

protected void initializePopulation(LocationsContainer locations, int size)
{
    for (int i = 0; i < size; i++)
    {
        population.add(generateIndividual(locations));
    }
}

private Tour generateIndividual(final LocationsContainer locations)
{
    LocationsContainer locationsToCreateTour = (LocationsContainer) locations.clone();
    Collections.shuffle(locationsToCreateTour);
    return new Tour(locationsToCreateTour);
}
```

Code Snippet 16 shows code to generate an individual

Collections.shuffle was used in this scenario to produce tours the locations being randomly inserted into the tour. From the official Java documentation (Oracle, 2015):

"Randomly permutes the specified list using a default source of randomness. All permutations occur with approximately equal likelihood."

The new initial population is now evaluated, in other words, evolved. Evolving a population consists of doing selection, crossover and mutation as per Figure 5.3. There are many ways to carry out these processes, and each of them has its advantages and disadvantages. The GA for this project applies Tournament Selection, a robust and commonly used mechanism that holds a tournament among a certain number of individuals and the individual with the highest fitness

level is selected as the winner of the tournament. Then, the winner is aggregated into the mating pool. Given that the mating pool is comprised of tournament winners, it is expected to have a higher average fitness than the average population fitness. The tournament size is selected randomly and has a direct impact on selection pressure that is the degree to which the fitter individuals are favored (L. Miller & E . Goldberg, 1995).

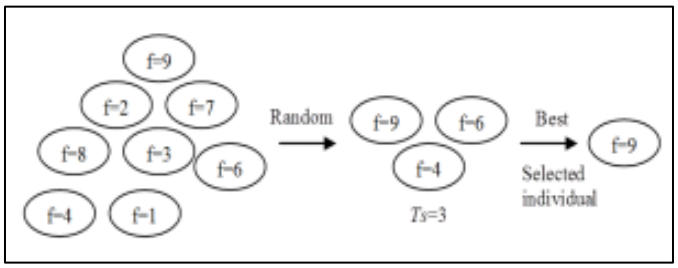


Figure 23 - Tournament Selection example **(Mohd Razali & Geraghty, 2011)**

The implementation of the Tournament Selection method is located in the GeneticAlgorithm class and the code can be observed below:

```
//This method does Tournament Selection
private Tour doSelection(Population population)
{
    Population tournament = new Population();

    for (int i = 0; i < tournamentSize; i++)
    {
        int randomTourIndex = (int) (Math.random() * population.getPopulationSize());
        tournament.addIndividual(population.getIndividual(randomTourIndex));
    }
    return tournament.getFittestIndividual();
}
```

Code Snippet 17 shows code to do Tournament Selection

The randomTourIndex variable stores a number generated randomly and it is used to select individuals from the population to add them into the tournament. Once the tournament is full the fittest individual in the tournament is selected as the winner and returned to the evaluation process for the next stage, crossover.

As mentioned above, after finishing the selection process the following step is to do crossover. However, major adjustments needed to be made to the TSPAlgorithm abstract class in order to accommodate the new GeneticAlgorithm class and its different requirements. These changes took the rest of the remaining time for this sprint, resulting in the genetic algorithm not being completed as planned, specifically the crossover and mutation steps. Therefore, these two tasks were marked as incomplete and moved on to the fifth sprint.

5.2.5 Sprint Five

Complete Genetic Algorithm

5.2.5.1 Finish implementation of the Genetic Algorithm

Task	**Status**
Implement Single Point crossover as the crossover method	Complete
Implement Swap mutation as the mutation method	Complete
Add elitism to genetic algorithm	Complete

Since the genetic algorithm could not be completed in the last sprint, the rest of the work was moved on to this sprint. After doing the selection, the next stage is crossover. As for the selection part, there are several ways to carry out this step; however, the method selected for this case was Single Point crossover. This crossover technique uses a single point that can be either generated randomly or set manually, to fragment the parents and then combines the two of them at the crossover point to produce a child or offspring (Umbarkar & Sheth, 2015).

Figure 23 shows an example of two parents being fragmented between fourth and fifth gene. After this, two offsprings are created combining the segments generated after setting the crossover point.

Parent 1:	1 0 1 0 | 1 0 0 1 0
Parent 2:	1 0 1 1 | 1 0 1 1 0
Offspring 1:	1 0 1 0 | 1 0 1 1 0
Offspring 2:	1 0 1 1 | 1 0 0 1 0

Figure 24 - Single Point crossover example **(Umbarkar & Sheth, 2015)**

The following code snippet shows the implementation of the single point crossover technique for this project.

```
private Tour doCrossover(Tour parentA, Tour parentB)
{
    int crossoverPointFactor = 2;
    Tour child = new Tour();

    int crossoverPointIndex = parentA.getNumberOfLocations() / crossoverPointFactor;

    //Copy individuals from the first parent till the crossover point
    for (int i = 0; i < crossoverPointIndex; i++)
    {
        child.addLocation(parentA.getLocation(i));
    }

    //The second parent is scanned and if an individual is not yet in the offspring, it is added
    for (int parentIndex = 0; parentIndex < parentB.getNumberOfLocations(); parentIndex++)
    {
        Location parentLocation = parentB.getLocation(parentIndex);
        boolean individualAlreadyPresent = false;

        for (int childIndex = 0; childIndex < child.getNumberOfLocations(); childIndex++)
        {
            Location childLocation = child.getLocation(childIndex);
            if ( parentLocation.getLocationId() == childLocation.getLocationId())
            {
                individualAlreadyPresent = true;
                break;
            }
        }

        if (!individualAlreadyPresent)
        {
            child.addLocation(parentLocation);
        }
    }

    return child;
}
```

Code Snippet 18 shows code to do Single Point crossover

As shown in the screenshot, this method requires two parents that are then used to generate a new child. The crossover point is manually set to be in the middle, therefore, the first half of the first parent is automatically copied into the new offspring. Then the second parent is fully

scanned to add the rest of the genes that have not been copied to the new offspring yet. Finally, the new child is returned to continue with the next step in the evaluation process, mutation.

The mutation method selected for this project was swap mutation, which consists of picking two genes randomly and swapping their positions (Eiben & Smith, 2003). Figure 24 shows a basic example of swap mutation where genes 2 and 5 are swapped.

Figure 25 - Swap mutation example **(Eiben & Smith, 2003)**

The following code snippet shows the implementation of the swap mutation technique for this project.

```
//This method does Swap Mutation
//Select two positions on the chromosome at random, and interchange the values
private Tour doMutation(Tour individual)
{
    int indexA = (int) (Math.random() * individual.getNumberOfLocations());
    int indexB;

    do
    {
        indexB = (int) (Math.random() * individual.getNumberOfLocations());
    } while (indexA == indexB);

    Location locationA = individual.getLocation(indexA);
    Location locationB = individual.getLocation(indexB);

    individual.setLocationAt(indexA, locationB);
    individual.setLocationAt(indexB, locationA);

    return individual;
}
```

Code Snippet 19 shows code to do Swap mutation

Elitism was put in place as well in an attempt to reduce the number of calls to the Google services and not reach the usage limit, as well as ensure the fittest individual of the population survived and was moved on to the next generation. With all these steps completed, the evaluation process was finished. Resulting in the genetic algorithm finished. The following screenshot displays the entire code for the evaluation process, where selection, crossover and mutation are being carried out.

```
private Population evaluatePopulation(Population population)
{
    int elitismFactor = 1;
    Population newPopulation = new Population();

    //Elitism is used to ensure that the best individual survives and
    //is copied into the next generation
    newPopulation.addIndividual(population.getFittestIndividual());


    for (int i = elitismFactor; i < population.getPopulationSize(); i++)
    {
        //Selection: Tournament Selection
        Tour parentA = doSelection(population);
        Tour parentB = doSelection(population);

        //Crossover: Single Point
        Tour child = doCrossover(parentA, parentB);

        //Mutation: Swap Mutation
        doMutation(child);

        newPopulation.addIndividual(child);
    }

    return newPopulation;
}
```

Code Snippet 20 shows code for the evaluation process

With the genetic algorithm finished, the code to process user selection was updated to use either the nearest neighbor or the genetic algorithm based on the number of locations. If the amount of locations is less than eight, then the nearest neighbor will be applied. Otherwise, the genetic algorithm is used. For this, the simple factory pattern was put in place to make sure the right algorithm is run. The code for this factory is displayed below.

```
public class AlgorithmFactory
{
    public static TSPAlgorithm createAlgorithmInstance(int numberOfLocations)
    {
        TSPAlgorithm algorithm = null;

        if (numberOfLocations < 8)
        {
            algorithm = new NearestNeighbor();
        }
        else
        {
            algorithm = new GeneticAlgorithm();
        }

        return algorithm;
    }
}
```

Code Snippet 21 shows AlgorithmFactory class

5.2.6 Sprint Six

Implement Brute Force algorithm

5.2.6.1 Add brute force algorithm to the project

Task	**Status**
Implement Brute Force algorithm	Complete
Add total distance covered by tour	Complete
Add field to specify starting node	Complete

The last algorithm to code was Brute Force, an exact solution that provides the actual shortest path, no approximations. However, it has a significant impact on the application's performance as it explores every possibility. The code for this algorithm is displayed in the code snippet below.

```
@Override
protected HashMap applyTSPAlgorithm(LocationsContainer locations, int start)
{
    //Find all locations
    ArrayList<Integer> locationIds = new ArrayList<>();
    for (int i = 0; i < locations.size(); ++i)
    {
        locationIds.add(locations.get(i).getLocationId());
    }
    ArrayList<Integer> route = new ArrayList<>();
    findAllRoutes(route, locationIds, locations);

    HashMap result = new HashMap();
    result.put("tour", shortestTour);
    result.put("distance", shortestDistance);

    return result;
}
```

Code Snippet 22 shows Brute Force implementation

The first step coded was to determine all the possible combinations. This is done by the findAllRoutes() method. Then, within this method there is a call to the processRoute() method, which calculates the distance after a tour is generated. The code snippet below shows the processRoute() method along with its helper method buildTour().

```
private void processRoute(ArrayList<Integer> route, LocationsContainer locations)
{
    int offset = 1;
    GoogleMapsDistanceMatrixClient client = new GoogleMapsDistanceMatrixClient();
    long distance = 0;
    for (int i = offset; i < locations.size(); i++)
    {
        distance += client.getDistanceBetweenTwoLocations(locations.get(route.get(i-1)), locations.get(route.get(i)));
    }
    System.out.println(route.toString() + " = Distance: " + distance);

    if (shortestDistance == 0)
    {
        shortestDistance = distance;
        buildTour(route, locations);
    }
    else if (distance < shortestDistance)
    {
        shortestDistance = distance;
        buildTour(route, locations);
    }
}

private void buildTour(ArrayList<Integer> route, LocationsContainer locations)
{
    LocationsContainer newTour = new LocationsContainer();
    for (int i : route)
    {
        newTour.add(locations.get(i));
    }
    shortestTour = newTour;
}
```

Code Snippet 203 shows code for processRoute() and buildTour() mehtods

This algorithm is used when the number of locations selected is less than five. It was tested several times producing successful results, and taking considerable time to finish as well.

Finally, a radio button was added to the GUI so that users can select a starting point for the tour if desired. If they decide to do so, the Nearest Neighbor algorithm will be used every time as this is the only algorithm that allows to specify a starting point.

Chapter 6: Findings & Analysis

6.1 Difficulties & Challenges Encountered

The implementation of this project was a challenge. The travelling salesman has been studied for the last decades and yet approximate solutions are still the best results for instances of this problem. TSP offers much complexity, especially when combined with little experience in the area, a wide set of available algorithms, and limited resources and time. Below are the issues encountered throughout the development of this project.

6.1.1 Numerous algorithms available

One difficulty was the vast number of algorithms available, which was overwhelming. With exact solutions, heuristics, meta-heuristics and approximation algorithms, it can be a challenge to decide which algorithms to implement for a project. They all have advantages and disadvantages, and work better in distinct scenarios; which may result in an extensive analysis of the instance given to determine what type of algorithms suits the problem better. This difficulty was overcome by selecting an easy-to-implement algorithm from each category.

6.1.2 Complexity in implementing a genetic algorithm

A genetic algorithm involves several steps, each step having different ways to be carried out. By doing a quick research on methods for selection, crossover and mutation, it is notorious the number of algorithms available that can be applied for each stage. Furthermore, there are also several methods for generating an initial population, which adds more complexity to the implementation, making it tedious and time-consuming. This difficulty was overcome by using the most common methods applied for each step.

6.1.3 Google Maps API usage limit

Probably the major challenge and block for this project. The free usage of the Google services is very limited in terms of the number of calls that can be made per day. The brute force and nearest neighbor algorithms work ok under this condition; however, a genetic algorithm involves generating new individuals (tours in this case) in every iteration, which results in the application having to issue several Google API calls to calculate the fitness level of each

individual (total distance to cover a tour). The stop criteria for the genetic algorithm develop for this project is 3 iterations only, and yet the limit is reached almost immediately. This causes the genetic algorithm to not be able to produce results. It only works with a few locations and a low number of iterations; however, these are not the ideal conditions for a genetic algorithm to work properly. The only way to overcome this difficulty is by obtaining a paid license to use the full capacity of the Google services. Prices and features can be seen on the official Google Maps API website under the section “Pricing & Plans”.

6.1.4 Generating sample data to test genetic algorithm

Since the Google Maps API usage limit was a known issue, the possibility of generating sample data to test the genetic algorithm was analyzed. Nevertheless, the amount of work involved was too much taken into account the remaining time until the deadline. This difficulty can be overcome by obtaining a paid license to use the full capacity of the Google services, avoiding the need of test data.

6.2 Conclusions & Recommendations

The research and implementation of this project aimed to evaluate the feasibility of combining the salesman problem with the Google Maps API services to produce a smart application for travelers who want to visit several locations in a distance-effective route. The findings, conclusions, and recommendations after completing this project are as follows.

6.2.1 Research Findings

Following the research undertaken on the travelling salesman problem, it is clear that there is no exact solution for every instance of it. The algorithm chosen is wholly dependent on the requirements of the given instance that needs to be processed. For small applications like this one, exact solutions and simple approximation algorithms such as the nearest neighbor appear to be the obvious choice. They offer a less complex approach and can be straightforward to understand conceptually and implement. On the other hand, for bigger applications genetic

algorithms seem to offer better results and performance. Therefore, if a given instance does not involve a great number of nodes, then by all means exact solutions or approximation algorithms should be used. Whereas if the number of nodes is big, a genetic algorithm would be a better candidate. It is true that the three options can produce results in both scenarios, however, the level of optimization generated by the brute force and nearest neighbor algorithms may be low in comparison with a genetic algorithm.

6.2.3 Implementation Findings

The development of this thesis allowed for a first-hand experience of the travelling salesman problem and the several Google maps API services. It is evident that the combination of these two concepts is possible and can help build smart applications for travelers in terms of route planning. The idea of exploring the different Google services was a beneficial and interesting concept for this project. All the information needed for the data processing was available through these services. Additionally, building the application under an agile methodology worked perfectly. This is evident in the implementation chapter as each algorithm had its own dedicated sprint, although the genetic algorithm needed to be extended to a second sprint due to some unforeseen issues with previous code developed. Having a genetic algorithm as part of this project was a big challenge, and a concept that took time to fully understand, especially the several stages that it requires.

It was also realized how easy it is to embed web pages into Java applications by using the JavaFX technology. The interaction between the two ends is simple and very customizable. There is no need for mounting a server to be able to load applications, and since the core is pure Java, it means the application can be run on every operating system that supports this technology.

6.2.3 Recommendations

Just as technology, the work on the TSP keeps consistently moving on, bringing new algorithms and shedding light on to what seems to be an unsolvable problem. So what does the future hold for the TSP? It does not appear that a definite algorithm will be found soon, after all, many

people have tried for many years with no success. However, more and more heuristics and approximation algorithms are being developed. Additionally, different techniques for genetic algorithms are constantly studied in order to improve results. Maybe one day someone will finally come up with the ultimate algorithm to solve every TSP instance in a reasonable manner. Only time will tell.

Based on the experience gained from the implementation, some of the recommendations for further development of this project are:

- Use Google maps features to draw tour on the map
- Use Google Places API to offer users attractions close to their destinations
- Have a database to store tours for future re-usage to improve performance
- Add JUnit tests to code in order to assure every functionality works as expected
- Refactor Genetic Algorithm code so that the selection, crossover and mutation steps are not limited to the methods implemented for this project but open to any other technique desired

References

Alabsi, F. & Naoum, R., 2012. Comparison of Selection Methods and Crossover Operations using Steady. *Journal of Emerging Trends in Computing and Information Sciences,* 3(7), pp. 1053-1058.

Apache Software Foundation, 2002-2015. *Apache Maven Project.* [Online]
Available at: https://maven.apache.org/what-is-maven.html
[Accessed 9th December 2015].

Chan, E. & Yang, Y., 2009. Shortest Path Tree Computation in Dynamic Graphs. *Computers, IEEE Transactions,* pp. 541 - 557.

Diaz-Gomez, P. A. & Hougen, D. F., 2007. *Initial Population for Genetic Algorithms: A Metric Approach.* Las Vegas, s.n.

Djojo, M. & Karyono, K., 2013. *Computational load analysis of Dijkstra, A*, and Floyd-Warshall algorithms in mesh network.* Jogjakarta, Robotics, Biomimetics, and Intelligent Computational Systems (ROBIONETICS), 2013 IEEE International Conference, pp. 104 - 108.

Djuknic, G. & Richton, R., 2002. Geolocation and assisted GPS. *Computer,* 34(2), pp. 123-125.

Eiben, A. & Smith, J. E., 2003. *Introduction to Evolutionary Computing.* 1st ed. Berlin: Springer-Verlag Berlin Heidelberg.

Foundation, T. A. S., 2002-2015. *what-is-maven.* [Online]
Available at: https://maven.apache.org/what-is-maven.html
[Accessed 7th December 2015].

Geisberger, R., 2008. *Contraction Hierarchies: Faster and Simpler,* Karlsruhe: Springer.

Gendreau, M. & Potvin, J.-Y., 2005. Metaheuristics in Combinatorial Optimization. *Annals of Operations Research,* 140(1-4), pp. 189-213.

GitHub, I., 2015. *GitHub features.* [Online]
Available at: https://github.com/features
[Accessed 9th December 2015].

Glovera, F., Gutinb, G., Yeoc, A. & Zverovichb, A., 2001. Construction heuristics for the asymmetric TSP. *European Journal of Operational Research,* 129(3), p. 555–568.

Google, 2010-2015. *AngularJS.* [Online]
Available at: https://docs.angularjs.org/guide/introduction
[Accessed 7th December 2015].

Google, 2010-2016. *Creating Custom Directives.* [Online]
Available at: https://docs.angularjs.org/guide/directive
[Accessed 26th January 2016].

Google, 2017. *Adding a Map with a Marker.* [Online]
Available at: https://developers.google.com/maps/documentation/javascript/adding-a-google-map
[Accessed 20 February 2017].

Google, 2017. *Directions API.* [Online]
Available at: https://developers.google.com/maps/documentation/directions/start
[Accessed 3 May 2017].

Google, 2017. *Distance Matrix API.* [Online]
Available at: https://developers.google.com/maps/documentation/distance-matrix/start
[Accessed 3 May 2017].

Google, 2017. *Distance Matrix API - Developer's Guide.* [Online]
Available at: https://developers.google.com/maps/documentation/distance-matrix/intro
[Accessed 5 March 2017].

Google, 2017. *Elevation API.* [Online]
Available at: https://developers.google.com/maps/documentation/elevation/start
[Accessed 3 May 2017].

Google, 2017. *Geocoding API.* [Online]
Available at: https://developers.google.com/maps/documentation/geocoding/start
[Accessed 3 May 2017].

Google, 2017. *Google Maps API - Pricing and Plans.* [Online]
Available at: https://developers.google.com/maps/pricing-and-plans/
[Accessed 3 May 2017].

Google, 2017. *Google Maps API: Documentation - Events.* [Online]
Available at: https://developers.google.com/maps/documentation/javascript/events
[Accessed 20 February 2017].

Google, 2017. *Google Maps APIs Web Services.* [Online]
Available at: https://developers.google.com/maps/web-services/overview
[Accessed 3 May 2017].

Google, 2017. *Introduccion to the Google Maps Roads API.* [Online]
Available at: https://developers.google.com/maps/documentation/roads/intro
[Accessed 3 May 2017].

Google, 2017. *Places API Web Service.* [Online]
Available at: https://developers.google.com/places/web-service/
[Accessed 3 May 2017].

Google, 2017. *Roads API.* [Online]
Available at: https://developers.google.com/maps/documentation/roads/intro
[Accessed 3 May 2017].

Google, 2017. *The Google Maps Geolocation API.* [Online]
Available at: https://developers.google.com/maps/documentation/geolocation/intro
[Accessed 3 May 2017].

Group, T. P. G. D., 1996-2015. *postgresql.* [Online]
Available at: http://www.postgresql.org/about/
[Accessed 7th December 2015].

Harrington, J. L., 2009. *Relational database design and Implementation.* 3rd ed. Burlington, MA: Morgan, Kaufmann.

Horstmann, C., 2012. *Big Java Early Objects.* Fifth Edition ed. New York: Wiley.

Jacobson, L. & Kanber, B., 2005. *Genetic Algorithms in Java Basics.* 1st ed. New York: Apress.

Jagadeesh, G. S. T. Q. K., 2002. Heuristic techniques for accelerating hierarchical routing on road networks. *Intelligent Transportation Systems, IEEE,* pp. 301-309.

Jayathilake, D. et al., 2012. *A study into the capabilities of NoSQL databases in handling a highly heterogeneous tree.* Beijing, IEEE, pp. 106 - 111.

Kammoun, S., Dramas, F., Oriola, B. & Jouffrais, C., 2010. *Route selection algorithm for Blind pedestrian.* Gyeonggi-do, Control Automation and Systems (ICCAS), 2010 International Conference.

Karimi, H. A., 1996. Real-time optimal route computation: a heuristic approach. *ITS J,* 3(2), pp. 111-127.

Király, A. & Abonyi, J., 2015. Redesign of the supply of mobile mechanics based on a novel genetic optimization algorithm using Google Maps API. *Engineering Applications of Artificial Intelligence,* Volume 38, pp. 122-130.

L. Applegate, D., E. Bixby, R., Chvátal, V. & J. Cook, W., 2007. *The Traveling Salesman Problem: A Computational Study.* 2nd ed. New Jersey: Princeton University Press.

L. Miller, B. & E . Goldberg, D., 1995. Genetic Algorithms, Tournament Selection and the Effects of Noise. *Complex Systems,* Volume 9, pp. 193- 212.

Laporte, G., 1992. The Traveling Salesman Problem: An overview of exact and approximate algorithms. *European Journal of Operational Research,* 59(2), pp. 231-247.

Liu, B., 1996. Intelligent route finding: combining knowledge, cases and an efficient search algorithm. *Conf. Artificial Intelligence, Cases and an Efficient Search Algorithm*, pp. 380-384.

Lupsa, L., Chiorean, I., Lupsa, R. & Neamtiu, L., 2010 . *Traveling Salesman Problem, Theory and Applications.* First ed. Rijeka: InTech.

McNeff, J., 2002. The global positioning system. *IEEE Transactions on Microwave Theory and Techniques,* 50(3), pp. 645 - 652.

Mohd Razali, N. & Geraghty, J., 2011. *Genetic Algorithm Performance with Different.* London, World Congress on Engineering.

Nejad, M., Mashayekhy, L. & Chinnam, R., 2012. *Effects of traffic network dynamics on hierarchical community-based representations of large road networks.* s.l., Intelligent Transportation Systems (ITSC), 2012 15th International IEEE Conference, pp. 1900 - 1905.

OpenStreetMap, 2015. *OpenStreetMap contributors.* [Online]
Available at: http://www.openstreetmap.org/about
[Accessed 9th December 2015].

Oracle, 2013. *JavaFX Overview.* [Online]
Available at: http://docs.oracle.com/javafx/2/overview/jfxpub-overview.htm
[Accessed 28 January 2017].

Oracle, 2015. *Class Collections.* [Online]
Available at: https://docs.oracle.com/javase/6/docs/api/java/util/Collections.html#shuffle(java.util.List)
[Accessed 3 May 2017].

Oracle, 2015. *JavaFX API - Class Region.* [Online]
Available at: https://docs.oracle.com/javase/8/javafx/api/javafx/scene/layout/Region.html
[Accessed 20 February 2017].

Pezzellaa, F., Morgantia, G. & Ciaschettib, G., 2007. A genetic algorithm for the Flexible Job-shop Scheduling Problem. *Computers & Operations Research,* 35(10), p. 3202–3212.

planning-policy-manual, N. Z. T. A., 2007. *Transit Planning Policy Manual version 1.* [Online]
Available at: https://www.nzta.govt.nz/assets/resources/planning-policy-manual/docs/planning-policy-manual-appendix-3A.pdf
[Accessed 7 oct 2015].

R. Jacob, M. V. M. a. K. N., 1998. *A computational study of routing algorithms for realistic transportation networks.* s.l.:s.n.

Rajagopalan, R., Mehrotra, K., Mohan, C. & Varshney, P., 2008. *Hierarchical path computation approach for large graphs,* s.l.: Aerospace and Electronic Systems, IEEE Transactions .

Redmond, E. & Wilson, J. R., 2012. *Seven Databases in Seven Weeks - A Guide to Modern Databases and the NoSQL Movement.* 1st ed. Dallas , Texas: The Pragmatic Bookshelf.

Redmond, E. & Wlson, J. R., 2012. In: J. Carter, ed. *Seven Databases in Seven Weeks.* Dallas, Texas: The Pragmatic Bookshelf.

Rego, C., Gamboa, D., Glover, F. & Osterman, C., 2011. Traveling salesman problem heuristics: Leading methods, implementations. *European Journal of Operational Research,* 211(3), p. 427–441.

Reinelt, G., 2003. *The Traveling Salesman: Computational Solutions for TSP Applications.* Berlin: Springer.

Sahalot, A. & Shrimali, S., 2014. A comparative study of brute force method, nearest neighbour and greedy algorthms to solve the traveling salesman problem. *IMPACT: International Journal of Research in Engineering & Technology,* 2(6), pp. 59-72.

Sghaier, M., Zgaya, H., Hammadi, S. & Tahon, C., 2010. *A distributed dijkstra's algorithm for the implementation of a Real Time Carpooling Service with an optimized aspect on siblings.* Funchal, Intelligent Transportation Systems (ITSC), 2010 13th International IEEE Conference, pp. 795 - 800.

Sheard, T., 2009. *Portland State University.* [Online]
Available at: http://web.cecs.pdx.edu/~sheard/course/Cs163/Doc/Graphs.html
[Accessed 11 November 2016].

Sierra, K., Freeman, E., Freeman, E. & Bates, B., 2004. *Head First Design Patterns: A Brain-Friendly Guide.* 1st ed. California: O'Reilly Media.

Soltys-Kulinicz, M., 2012. *Introduction to the Analysis of Algorithms (2ND EDITION).* 2nd Edition ed. s.l.:World Scientific Publishing Co..

Svennerberg, G., 2010. *Beginning Google Maps API 3.* 2nd ed. New York: Apress.

Takashi UCHIDA, Y. I. a. M. N., 1994. *Panel Survey nN Drivrs' Route Choice Behaviour Under Travel Time Information,* Yokohama: Vehicle Navigation and Information Systems Conference.

Team, T. Y., 2015. *TOOL FOR MODERN WEBAPPS.* [Online]
Available at: http://yeoman.io/
[Accessed 9th December 2015].

Umbarkar, A. & Sheth, P., 2015. CROSSOVER OPERATORS IN GENETIC ALGORITHMS: A REVIEW. *ICTACT JOURNAL ON SOFT COMPUTING,* 6(1), pp. 1083-1092.

Wang, Y., 2014. *A Nearest Neighbor Method with a Frequency Graph for Traveling Salesman Problem.* Hangzhou, IEEE.

White, J. L. B. a. C. C., January 1998. A Heuristic Search Algorithm for. *IEEE TRANSACTIONS ON SYSTEMS, MAN, AND CYBERNETICS—PART A,* 28(1), p. 131.

Wikistack, 2016. *Wikistack.* [Online]
Available at: http://wikistack.com/traveling-salesman-problem-brute-force-and-dynamic-programming/
[Accessed 11 November 2016].

Wolffelaar, J. v., 2010. *Highway Node Routing: increasing flexibility and putting it into practice,* Utrecht,: Utrecht University.

Xiao, J.-X. & Lu, F.-L., 2010. *An improvement of the shortest path algorithm based on Dijkstra algorithm.* Singapore, Computer and Automation Engineering (ICCAE), 2010 The 2nd International Conference, pp. 383 - 385.

Yanwu, Z., Yongqiang, S., Na, Z. & Meng, Z., Aug 2011. *Comparing Study of Route Planning Algorithms Based on Hierarchical Strategy,* s.l.: IEEE Conference Publications.

Zhou, L., 2012. *Evaluating road selectivity of urban-trip based on dynamic betweenness centrality.* Hong Kong, Geoinformatics (GEOINFORMATICS), 2012 20th International Conference.