UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Route mapping of a city using OpenStreetMap and planning for optimal traversal

Diploma Thesis

Giannikoglou Kalliopi

Supervisor: Thanos Georgios

June 2024

UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Route mapping of a city using OpenStreetMap and planning for optimal traversal

Diploma Thesis

Giannikoglou Kalliopi

Supervisor: Thanos Georgios

June 2024

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ
ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Χαρτογράφηση διαδρομών μιας πόλης με χρήση του OpenStreetMap και σχεδιασμός βέλτιστων διελεύσεων

Διπλωματική Εργασία

Γιαννίκογλου Καλλιόπη

Επιβλέπων: Θάνος Γεώργιος

Ιούνιος 2024

Approved by the Examination Committee:

Supervisor **Thanos Georgios**
Laboratory Teaching Staff member, Department of Electrical and Computer Engineering, University of Thessaly

Member **Tousidou Eleni**
Laboratory Teaching Staff member, Department of Electrical and Computer Engineering, University of Thessaly

Member **Fevgas Athanasios**
Laboratory Teaching Staff member, Department of Electrical and Computer Engineering, University of Thessaly

Acknowledgements

Taking this opportunity, I would like to express my sincere appreciation and gratitude to my supervisor, Dr. Thanos Georgios, for his constant help and guidance through this learning journey. It could not have happened without his contribution, and I deeply thank him for that.

I would also like to thank Dr. Tousidou Eleni for her support in my early academic steps and her participation in the Examination Committee of this thesis. Additionally, I would like to thank Dr. Fevgas Athanasios, for eagerly accepting a position in the Examination Committee.

Moreover, I owe a special thanks to my parents, Angeliki and Vasilis, and my brother, Kostas, who support me unconditionally and have borne with me all these years.

Last but not least, I want to express my love and gratitude to all my friends, both those who have been by my side from the beginning and those I have made along the way. Nothing would be possible without them.

DISCLAIMER ON ACADEMIC ETHICS
AND INTELLECTUAL PROPERTY RIGHTS

«Being fully aware of the implications of copyright laws, I expressly state that this diploma thesis, as well as the electronic files and source codes developed or modified in the course of this thesis, are solely the product of my personal work and do not infringe any rights of intellectual property, personality and personal data of third parties, do not contain work / contributions of third parties for which the permission of the authors / beneficiaries is required and are not a product of partial or complete plagiarism, while the sources used are limited to the bibliographic references only and meet the rules of scientific citing. The points where I have used ideas, text, files and / or sources of other authors are clearly mentioned in the text with the appropriate citation and the relevant complete reference is included in the bibliographic references section. I also declare that the results of the work have not been used to obtain another degree. I fully, individually and personally undertake all legal and administrative consequences that may arise in the event that it is proven, in the course of time, that this thesis or part of it does not belong to me because it is a product of plagiarism».

The declarant

Giannikoglou Kalliopi

Diploma Thesis

Route mapping of a city using OpenStreetMap and planning for optimal traversal

Giannikoglou Kalliopi

Abstract

In the post COVID-19 era, the percentage of online markets has been continuously increasing, creating a greater demand for delivery and distribution services. Even though the necessity is considerable, the existing solutions are not addressing the problem sufficiently, as the essence of the problem is extremely complex and multifactorial.

That was the motivation behind the present diploma thesis, where we attempted to give a solution to this complex problem. We defined the problem as the route mapping of a city using OpenStreetMap service. Given an amount of packages to be distributed and the number of available agents, we planned the distribution and the optimal traversal of all the locations for each agent. This procedure is also supported by a single-page web app to be user-friendly and utilizable for a distribution company.

More specifically, we focused on one city at a time. Given an OpenStreetMap (.osm) file, we encoded all the existing streets, roundabouts, buildings, and points of interest, such as natural features, facilities or recreational areas. The next step was to define all the locations we wanted to include in our traversals, including the initial positions of the available agents. Once this was complete, we performed calculations to distribute the locations fairly among the agents. Using Nearest Neighbor Heuristic algorithm we searched for the most efficient order of traversal and then, using A* algorithm, we found the optimal traversal for each agent. The result of these calculations is maps with all the paths each agent has to follow to traverse all the locations.

Keywords:

Route Mapping, Optimal Traversal, Travelling Salesman Problem, A* Search, Nearest Neighbor Heuristic Algorithm, OpenStreetMap, React.js, Node.js

Διπλωματική Εργασία

Χαρτογράφηση διαδρομών μιας πόλης με χρήση του OpenStreetMap και σχεδιασμός βέλτιστων διελεύσεων

Γιαννίκογλου Καλλιόπη

Περίληψη

Στη μετά COVID-19 εποχή, οι διαδικτυακές αγορές αυξάνονται συνεχώς, δημιουργώντας μεγαλύτερη ζήτηση για υπηρεσίες παράδοσης και διανομής. Ενώ η ανάγκη είναι αξιοπρόσεκτη, οι υπάρχουσες λύσεις δεν αντιμετωπίζουν επαρκώς το πρόβλημα, διότι είναι σύνθετο και πολυπαραγοντικό.

Αυτό ήταν το κίνητρο πίσω από την παρούσα διπλωματική εργασία, όπου επιχειρήσαμε να προτείνουμε μια λύση. Ορίσαμε το πρόβλημα ως τη χαρτογράφιση μιας πόλης με τη χρήση του OpenStreetMap και σχεδιάσαμε την βέλτιστη διάσχιση για κάθε διανομέα, λαμβάνοντας υπόψη τον αριθμό διαθέσιμων διανομέων και τα πακέτα που πρέπει να παραδοθούν. Η διαδικασία υποστηρίζεται από μια διαδικτυακή εφαρμογή, ώστε να είναι φιλική προς τον χρήστη και αξιοποιήσιμη από μια εταιρεία διανομής.

Επικεντρωθήκαμε σε μία πόλη κάθε φορά και, παίρνοντας ένα αρχείο OpenStreetMap (.osm), κωδικοποιήσαμε δρόμους, κόμβους, κτήρια και σημεία ενδιαφέροντος. Το επόμενο βήμα ήταν να πάρουμε τις τοποθεσίες παραλαβής και παράδοσης των πακέτων και τις αρχικές θέσεις των διανομέων. Υπολογίσαμε τη δικαιότερη κατανομή των παραδόσεων ανάμεσα στους διανομείς και, χρησιμοποιώντας τον ευρηστικό αλγόριθμο Κοντινότερου Γείτονα, αναζητήσαμε την καλύτερη σειρά διάσχισης των τοποθεσιών. Έπειτα, με τον αλγόριθμο A*, βρήκαμε την βέλτιστη διάσχιση για κάθε διανομέα. Το αποτέλεσμα είναι χάρτες με τα μονοπάτια κάθε διανομέα για την εξυπηρέτηση όλων των παραδόσεων.

Λέξεις-κλειδιά:

Χαρτογράφιση Διαδρομής, Βέλτιστη Διάσχιση, Το πρόβλημα του Πλανόδιου Πωλητή, Αναζήτηση A*, Ευρηστικός Αλγόριθμος Κοντινότερου Γείτονα, OpenStreetMap, React.js, Node.js

Table of contents

List of figures

Abbreviations

OSM	OpenStreetMap
JS	Javascript
API	Application Programming Interface
K-D Trees	K-Dimensional Trees
IDE	Integrated development environment
RMNode	Route Model Node
DOM	Document Object Model
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation

Chapter 1

Introduction

As we experience the vast growth of technology, more and more aspects of our daily lives are digitized, offering us ever-increasing convenience. A great example of the transformations that are highly adopted by the consumers is online shopping and e-commerce. We now have the ability to purchase a variety of goods, from food and cloths, to books and flowers, via a variety of web and mobile applications.

To meet this growing demand, many delivery companies are coming up with innovative solutions to serve these needs, from the traditional door-to-door approach, to more creative large-scale options. An instance of the latter is the central distribution places where the customers pick up their personalized packages by themselves.

All the above approaches can be deducted to the common problem of task distribution and search for optimal routing. This is the purpose of this diploma thesis, aiming to give a practical and competitive solution that addresses this complex problem effectively.

1.1 Main Objective

The primary objective of the present diploma thesis is to address the complex issue of optimal route mapping within a city, considering multiple delivery agents assigned with packages to distribute. The process will be supported by a web application that will aid the user interaction with the system and visualize the final results.

To make a study on this complex matter, we have to resolve several sub-problems. These problems include finding the minimum path between two points inside the city and determining an efficient order to traverse multiple destination points. We also have to develop an algorithm

that handles the task distribution problem, which is the assignment of packages to the available delivery agents.

After this process is complete, we will build a user-friendly web application, to integrate and demonstrate the results of our algorithms. We do not intent to develop an application that will be deployed in the real world, but for a simple and functional web service with limited features, presenting our algorithmic findings.

1.1.1 Relevant Work

Before starting our work on the problem we want to address, we made some research for similar applications that are already used in the real world. We could consider that applications, such as Circuit[1] and Maposcope[2], are resolving the same problems in terms of making a route planning and focusing on handling multiple stops and optimize their order of traversal.

More specifically, Circuit and Maposcope intend to provide multi destination route planning. The user, who might be a delivery agent, inserts all the destinations they want to visit. There are some features that allow to prioritize some destinations over others or give a time frame through some of the places that must be visited. After all the locations are inserted in the system, there is the option of optimization, where the system returns the most efficient path to visit all those destinations. Both applications are based on the Google Maps App and the results are projected in the Google Maps format.

1.1.2 Contribution

Having defined the problem we aim to address in this thesis and reviewed current solution by the corresponding applications at the moment, we now present our contributions to the matter:

1. Points as pairs: Our solution introduces a route mapping approach that treats visiting points as sources and destinations of packages, rather than one-dimensional points that could be accessed in any order. Unlike most of the existing options that are focusing on optimizing a path, similar to solving the Travel Salesman problem, we approach all the points in pairs, focusing in the real packages delivery issue.

2. Package assignment to delivery agents: Our study includes the assignment of packages to delivery agents. Unlike most of the applications that are offering route mapping for an

individual and a single ride, we are addressing the issue closer to the root. We propose a solution that includes the task distribution and the load balance management.

3. Google Maps API: The web application will integrate the Google Maps API[3], which is used by the vast majority of people, making the user interface more familiar. Additionally, it will be a solid foundation for future expansion as a real world application.

1.2 Thesis Structure

This section is a roadmap of the structure of the thesis. We will present how thesis is articulated, to ensure better comprehension and to give a first idea on how our work is structured.

Starting with Chapter 1, we provide an introduction to the aim of this thesis, followed by the main objectives of our work, giving a brief overview of the addressed issues. Then we shortly present some relevant work and in Contribution subsection we point what we are offering in this field of study.

Then, Chapter 2 presents all the developed and integrated algorithms and explaining thoroughly their important roles in this project. It is where all the algorithmic process is described, enriched with pseudo-codes for the most important implementations.

In Chapter 3 we present the development of the web application. We are doing a thorough reference to the tools and technologies used for the Frontend and the Backend of the application, and we also highlight the most important parts of our development process.

Chapter 4, gives a high level navigation on the web application, by presenting the user interface and the important components included. Through the navigation, a usage example is presented, where the results of our algorithm are also showcased.

Lastly, in Chapter 5, our research is summarized, our results and findings are presented through a critical analysis and there are some proposed directions for potential future work.

Chapter 2

Route Mapping and Optimal Traversal

2.1 Introduction

In this Chapter we will analyze thoroughly all the algorithmic process we followed to address the complex issues of package distribution, optimal traversal order and route mapping within the city.

During the development process, we organized these problems in 4 phases, based on which we will structure this Chapter:

- **Phase 1**: Route mapping from one source to one destination point, using A* Search
- **Phase 2**: Route mapping from one source to multiple destinations and search for optimal order of traversal, using the Nearest Neighbor approach
- **Phase 3**: Route mapping from multiple sources to multiple destinations, developing a variation of Nearest Neighbor Heuristic algorithm that treat points as start and end points.
- **Phase 4**: Integrate the above steps for multiple delivery agents and development of an algorithm to perform package distribution, using K-D Trees

2.1.1 Technology and Tools Used

The code for the described phases was developed in Java programming language[4] using the Intellij IDE from JetBrains [5]. Additionally, we used Google Maps API [3] in order to generate all the images (.png files) of the maps with the final paths on them. The last very

important tool to be mentioned is Github [6], where all the source files were uploaded during the implementation steps.

Now, we will explain more analytically what these tools are and why they were chosen in our project.

- **Java Programming Language:** As mentioned, all the source code of the algorithmic part was written in Java[4]. It was chosen since it is a widely-used language for object-oriented programming, as it is fast, robust, can perform complex calculations and thus, it is considered highly reliable in real world applications. Additionally it can be used as the server-side of a web service, which was extremely important in our case.

- **Intellij IDE:** Having decided to use Java, we chose Intellij Integrated Development Environment (IDE)[5], distributed by JetBrains. It is very popular among developers since it offers useful features, like intelligent code editor that makes the development and debugging process easier. Another very important feature is the easy remote connection with Github, saving a lot of time to the developer and making the coding process more efficient.

- **Github:** The next tool we used was Github, the web platform used for Git, which allows remote collaboration, online storage of the files and it keeps a history with all the commits, allowing the developer to backtrack previous versions of the code. It also provides multiple tools for organization of the work, such as ”Projects”, workflows and issues.

  We strongly encourage looking in the according Github Repositories for a deeper comprehension of the code, as in the thesis only the most important parts of the development are highlighted. The code for the three parts are publicly available in different repositories for clarity. The three repositories are divided in algorithmic part [7], Backend in Spring Boot [8] and Frontend in React [9].

- **Google Maps API:** The last widely used tools we integrated in our project is Google Maps Application Programming Interface (API) [3], which allows the user to embed Google Maps inside their application. After generating a unique API key, the developer has access to multiple features, that offer a variety of functionalities. We used the API to generate the images (as .png files): we inserted all the points in the system, and paved the final path with a blue line and added markers for the important points. The source points

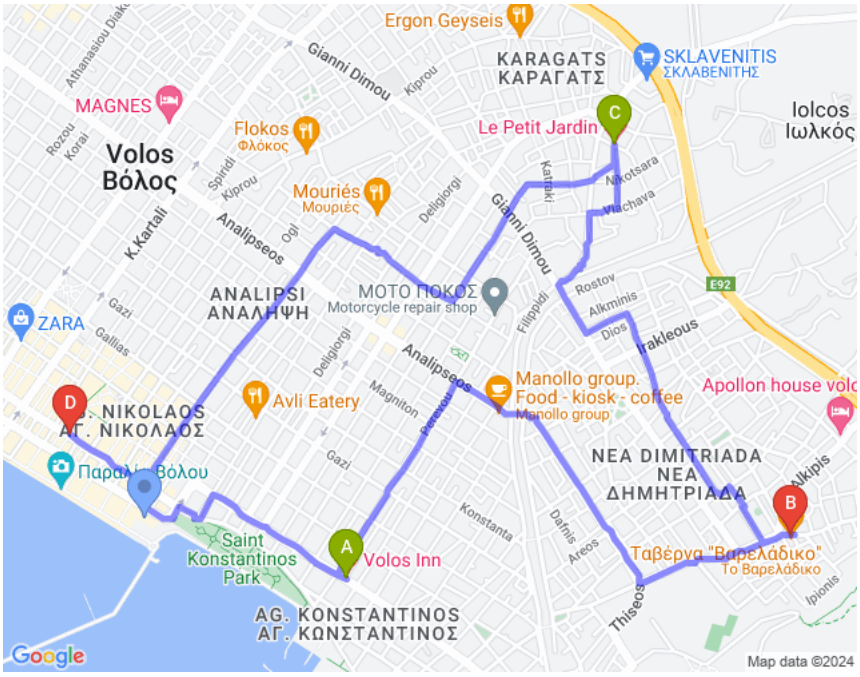


Figure 2.1: Example of Google Map API generated images

were marked in green, destination points in red and agents starting positions in blue. The markers also included labels with the letters of the alphabet to show the order of traversal for each point. An example of the generated images is shown in Figure2.1.

2.1.2 Java Classes and Class Diagram

Before proceeding to analyzing the 4 phases of development of the algorithmic part, it would be highly beneficial to look at the bigger picture of the Object Oriented Model of this project. This would make it easier to follow the thinking process and the connection between the elements of the project.

The most common way to present that is through a UML Diagram, which is an overview of the Java Classes and is shown in Figure 2.2. It was created with the online tool Visual Paradigm[10].

As demonstrated in Figure2.2, all the decoded map components are stored in *Node*, *Way* and *Road* Classes. The decoding is implemented in *Model* Class, which handles the XML input file. There is also the *RMNode* Class that inherits *Node* Class where the implemented methods handle relations between the Nodes of the map, such as finding Neighbors (close nodes), which is an important step for the A* Search implementation.

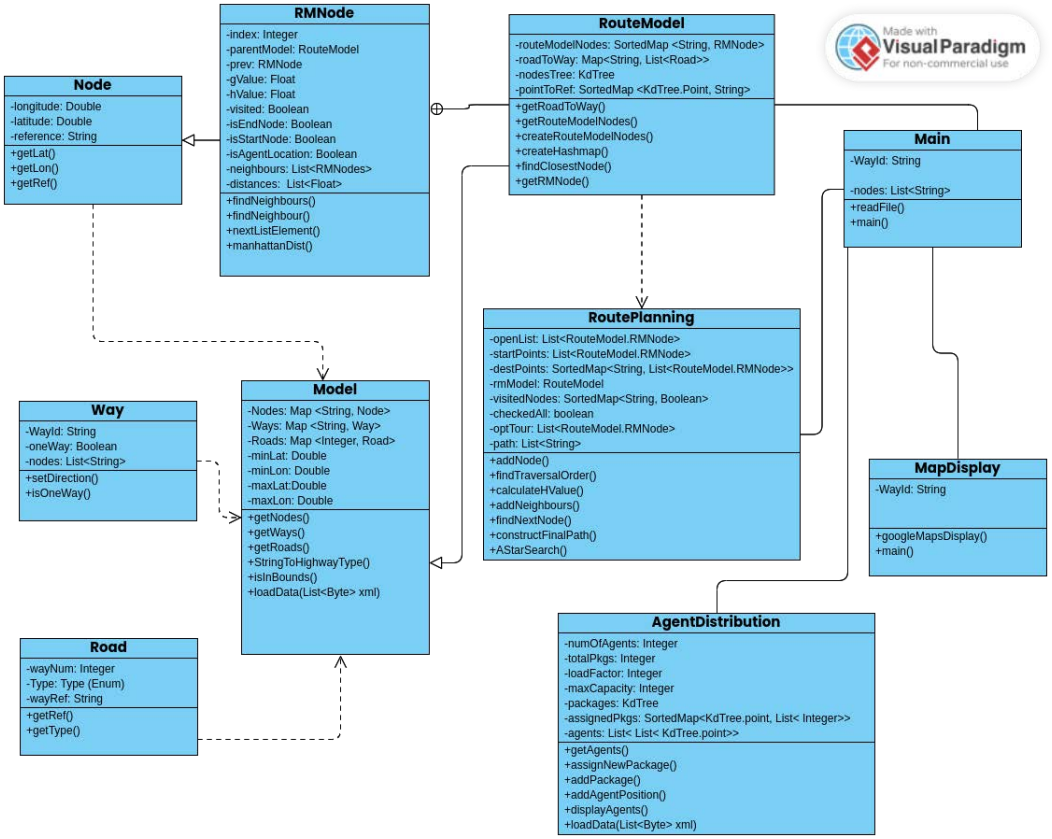


Figure 2.2: UML Diagram of Java Classes

The *RouteModel* Class contains the *RMNode* Class, and is performing a supportive role around the connection of Route Model Nodes. For instance, in this class a hash map is created which connects all the Ways that consist a Road and additionally all the Route Model Nodes are inserted in a K-D Tree that accelerates search of a specific node. The latter is fundamental in the execution time of our calculation an it will be analyzed further in the next sections.

The next class is the *RoutePlanning* Class which connects with *Main* Class, as it receives the source and the destination points, does all the processing needed for the implementation of A* Search algorithm, and then applies it to find and construct the final path. This class was changing significantly through the phases of development, as more complex processes were required. The alterations will be thoroughly explained in the according parts.

The *AgentDistribution* Class is one of the last Classes developed, since it was used to perform the task distribution amongst the agents. This was needed in the last phase of development where we added the agents parameter in our calculations.

The last class to be mentioned is the *MapDisplay* class which was used to generate the final map images with the paved path and colored markers to show the starting and finishing points

of the packages. In the last phase of the implementation, we also added markers for the positions of the agents. This representation was performed using Google Maps API [3] and it was a very helpful tool through the development process, as it was used for testing our results.

2.2 Phase 1: Route Mapping between two points

Efficient route mapping between two points within the map is a fundamental task for our project. This is where we started the development process and this is the primary topic in this section. The very first step was to handle the map, which is an OSM (.osm) file, that we transformed in useful and agile data structures, such as K-Dimensional Trees. We then defined all the Java Classes of our project and implemented the necessary methods, that allowed us to integrate A* Search from point A to point B.

2.2.1 Decoding of OpenStreetMap file

Starting with OpenStreetMap [11], we had to export the map of the city we wanted to work with. Our choice was the city of Volos and thus we defined our map bounds to be:

```
<bounds minlat="39.349" minlon="22.9"
        maxlat="39.388" maxlon="22.982"/>
```

After we exported our map file we had to find the appropriate data structures to save the useful information, which was the nodes, the ways and the roads. It is important to mention that it is impossible to get all the coordinates of a map, even if the area is finite, since the points on the map are infinite. This is why OSM files provide nodes with sufficiently small distances, that represent all the junctions of roads, without losing useful information. The file also contains information about all the points of interest in the map, such as the port, the buildings and the playgrounds, that are considered obstacles (or blocked ways) in route mapping tasks .

For the parsing of the OSM file, which is an XML file, we used Jsoup Java library [12], which is a Java HTML and XML parser. The Java structure we decided to use, in order to store the file information, was Java built-in Interface Map<K, V> [13] from java.util library, which connects one key with one value.

So, we used multiple Map Structures to store the Nodes, the Roads and the Ways of the OSM file. Additionally, for the roads we had to store information about their type (Highway,

Secondary, Foot way etc.) and for the Ways we had to keep in mind if their one or two way.

After this process was complete, we had decoded all the OSM file information and we could proceed in the next steps of our development.

2.2.2 K-Dimensional Trees

A very important decision we had to take as we begun the development journey, was which data structure would be used to store the map Nodes. There was a plethora of nodes that had to be accessed very often and the search amongst them had to be as fast as possible. All the above conditions were completed by K-Dimensional Trees (K-D Trees).

The K-D Trees are balanced, binary trees where every entry represents one point in the K-Dimensional Space. In our case, maps are represented, so every tree node has 2-dimensions (x, y) that represent latitude and longitude. The general idea of this tree is to partition the space in the according dimensions, so every non-leaf entry is tracing a line that divides the space in two, based on the defined axes. Additionally, the rows of the tree represent each of the axes alternately, so the first row is defining a line parallel to the x-axis, the second row a line parallel to the y-axis and then again the third to the x-axis. The above segmentation is shown in Figure2.3.

This structure is very efficient, as it is a balanced tree that allows an average complexity of O(logn) for insertion and for search. This gives us the opportunity to search multiple nodes with a multi-dimensional key in very short time.

The K-D trees are not a built-in structure of Java, so we used the code from the ‘java-algorithms-implementation‘ repository by Justin Wetherell[14]. The code is used under the terms of the Apache License, which allows for reuse and modification with proper attribution to the original author. The full text of the Apache License can be found here.

The above implementation includes a method that finds nearest Neighbor in the tree, by calculating the distances of the points in the 2-D space. This method was widely used in our implementation as it could get any coordinate from the map as input and return the closest existing node in our tree, allowing us to work only with valid nodes.

2.2.3 A* Search

Having discussed about all the prerequisites for Phase 1, we can now analyze the A* Search (A-star Search). This widely popular path-finding algorithm is ideal for our objectives for this

Fig.2.3 Source: `https://www.baeldung.com/cs/k-d-trees`

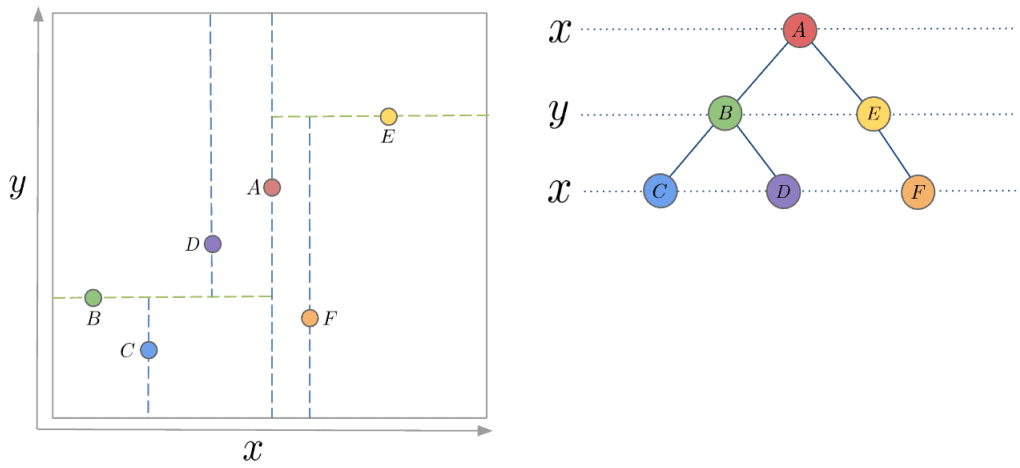


Figure 2.3: Example of K-D Tree

phase. We want to find the optimal path to get from one point A to a second point B, within a map.

The A* Search algorithm is considered an evolution, or an extension, of the famous Dijkstra's algorithm. All this popularity is due to its heuristic nature, used to decide on the next steps, overcoming the performance of Dijkstra. It also overcomes the efficiency of the majority of conventional search algorithms since it is "smarter" and approximates the destination, avoiding all the obstacles a real-life map may have, such as buildings.

The way all these promising results are achieved is by deciding the next moves through the **f-value**. This value is the sum of two values:

1. The **g-value:** all the cost to get to the present node, that includes the weight of all the previous points.

2. The **h-value:** all the approximated cost to reach the final destination. This is where heuristics are used. We do not actually calculate the real distance to get there, but an approximation of this, using Manhattan Distance (L1 norm)[15].

It was decided to use Manhattan distance, as it was giving similar results with the Euclidean distance (L2 norm)[16], without burdening the system with additional computational cost. It was a computation that would be used multiple times, so we needed a lightweight solution.

We note that the way the two parameters are defined is what give "brains" to the A* Search, as it provides every time sufficient knowledge for the past and the future of the current position.

Another important characteristic of this algorithm is which movements are allowed and thus, which nodes are considered Neighbors. The movement is happening as moving in a grid,

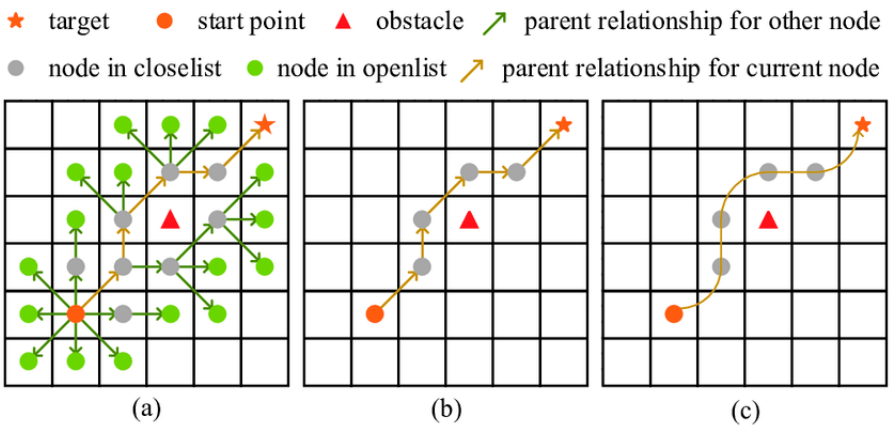


Figure 2.4: Example of A* Search movement

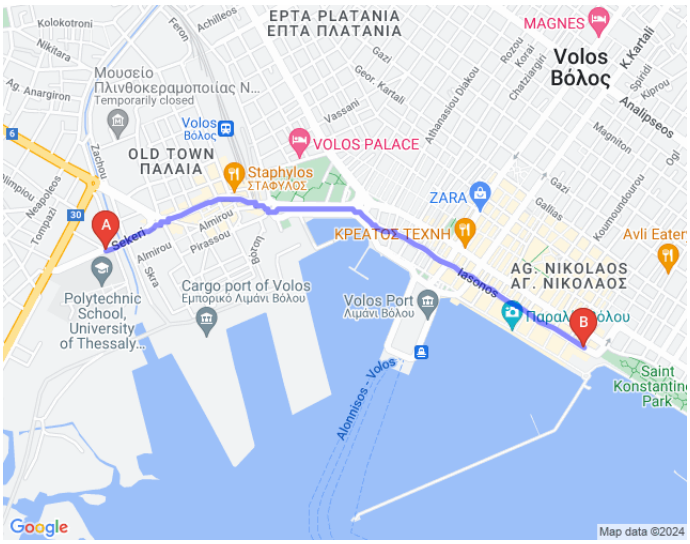


Figure 2.5: Execution Results of Phase 1

so every time our current destination is in the center of a 9-cell grid and the 8 surrounding cells are Neighbors and candidate future positions. There is also the capability to detect obstacles, for example buildings of a city, and go around them. This procedure is visualized in Figure 2.4, clarifying the available movements for any position.

In our implementation, A* Search was used in the original form, with no alterations. The pseudo-code of A* search is available in Appendix A.1. Additionally, we provide an indicative example of the execution at this point, as shown in Figure2.5, where we expecting routing between points A and B.

Fig.2.4 Source: `https://www.researchgate.net/figure/The-grid-graph-at-a-certain-step-in-the-search-process-of-the-A-star-algorithm-a-shows_fig1_357205674`

2.3 Phase 2: Multiple destinations and Optimal Traversal Order

The next phase of our implementation was to achieve routing between multiple points, having one designated start. To achieve this we expanded the code developed in Phase 1, which found the optimal path between two points, and proceeded with searching the most efficient way to traverse the multiple destinations.

This problem, is widely used in the real world, for instance in a post office, where all the packages and letters have one common source and the postman (or delivery agent, in our case) has to visit all the destinations to distribute them. The challenging part in this case study is in which order the destinations will be visited. After finding that order of traversal, we just apply the A* Search for all the points, in pairs.

2.3.1 Nearest Neighbor Heuristic Algorithm

Searching the most efficient traversal order, visiting every destination exactly once, is the very common NP-hard Traveling Salesman Problem (TSP)[17]. It is an NP-Hard problem as, in the worst-case, the execution time is increasing sharply and the computational cost explodes and there is no polynomial as an upper bound.

Having explained the tricky parts of the conventional TSP, it is clear that searching exhaustively for a route mapping inside a city, with various source and destination points, would be extremely complex. This is why we decided to follow a heuristic approach, which would make a reliable approximation in a short period of time. The algorithm we followed is the Nearest Neighbor Heuristic approach.

The Nearest Neighbor Heuristic algorithm is a greedy algorithm and the main is to choose the Neighbor with the smaller distance from current node. The concept is pretty simple, as most of the greedy approaches, but always gives a solution. In our case, this approach is well-suited since it has a simplicity in the implementation, making it an excellent basis for structuring the third phase of our implementation. We needed a flexible algorithm, that could allow us to make the alterations needed to fit our objectives.

Additionally, since all the algorithm is based in calculating distances to make the next choice, it is a solution that requires minimum resources, the execution time is very small and there is no need for extra space. This was highly beneficial in our case, as we needed this search

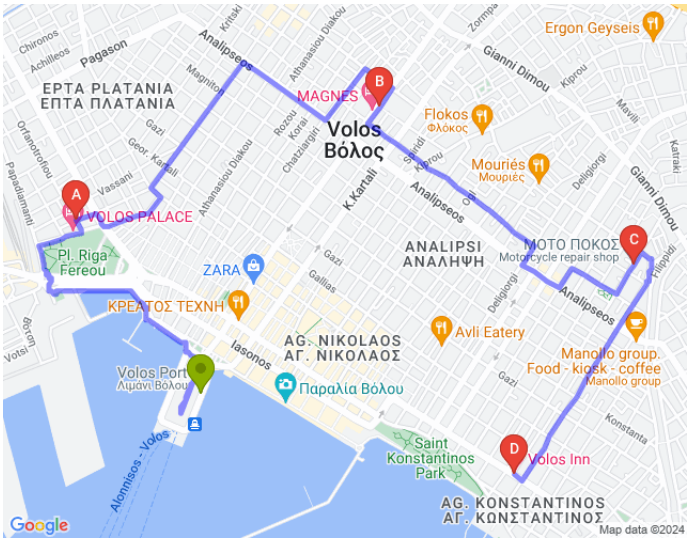


Figure 2.6: Execution Results of Phase 2

to be as fast as possible, as it is applied as many times as the number of available agents. A validation of the above statements comes from the following publication [18], where among plenty of heuristic approaches on the TSP problem, Nearest Neighbor Heuristic converges the fastest in the least possible iterations.

This heuristic approach always gives a result, but sometimes it is not the optimal. One of the reasons this happens is because of the randomness of the starting point, and changing the starting point, returns different results. Additionally there are no mechanisms that predict the ”future”, as we mentioned for the A* search. In our case, our search always starts from the initial position of the delivery agent, so randomness is minimized and even though the results are not always the optimal, they are reasonable.

The efficiency of the algorithm has been studied a lot, since Nearest Neighbor approach is very famous in TSP solutions and has plenty of variations. There are many publications [19] [18] [20] that come up to the result that, even though it does not always finds the optimal path, it is very stable and it always converges at least to the average solution. Mathematically it has been proven[19] that in worst case it is $(\frac{1}{2}logn + \frac{1}{2})OPTIMAL$, proving the previous statement.

The pseudo-code for Nearest Neighbor Heuristic algorithm, with our small alteration, can be found in Appendix A.2. We used again Manhattan Distance to calculate the distances between the nodes, since it is faster than Euclidean and the results are very similar.

An indicative example of the results of the execution at this phase is the shown in Figure2.6 where the green marker is our starting point and the red markers are all the destination points. The labels inside the markers indicate the order of traversal.

2.4 Phase 3: Route Mapping for multiple sources and destinations

In the current phase we wanted to expand our problem even more by adding multiple sources for the packages. So now, we do not have one start and many destinations, but all the packages come in pairs of source and destination. In the real world, this would apply more to a delivery system that is distributing packages from different stores or restaurants, where the source is not a central station where all the packages are starting, but the delivery agent has to pick the packages from all around the city.

This created a different case study from the traditional Travel Salesman Problem as there were some restrictions to the visiting order of the nodes. The restrictions were about the availability of the nodes, since a destination node could not be accessed before the pair starting node was visited. This needed a more complex solution from our side.

The initial idea was to keep the heuristic and greedy approach of the Nearest Neighbor Algorithm, but this time, in the beginning, we can only visit the starting points. So we calculate the distances of all the source points and we choose the closest to the current node. Once the node is visited, we find their pair (the according destination node) and add it to the list of available nodes.

Even though the main idea does not seem far from what we have already implemented for the previous phase, this case had many challenges during the implementation process. It was the first time we had to treat the nodes differently, based on whether they were source or destination points. From that point, we had to take decisions based on the real-life on how we would handle some situations.

For example, we had to decide how to treat the cases of two or more packages with the same source or the same destination. The common source, was not so challenging but the same destination created a dilemma on whether we should be able to visit a shared destination when one of their sources is visited, or we should wait until all the sources were visited and then go there once.

According to the Traveling Salesman Problem, each destination must be visited exactly once, but in real world, it is sometimes more beneficial to deliver the packages when you get them, as it helps maintaining a better space management system, but also improves the time management.

2.4.1 Order of Traversal Algorithm

Having described the purpose of this phase of the implementation and having mentioned the challenges we had to overcome, we will now present the structured idea and the pseudo-code for our implementation.

The first thing we have to explain is the data structures chosen:

- A List<SourceNodes> [21] to save our source points. So there was a list with all the source points, where duplicates were allowed.

- A SortedMap<SourceNode, List<DestNode> > [22] to save our destination points. The above data structure means it is a Sorted Map, so we have unique keys K that are mapping values V and we keep them sorted by key, for faster searching. The key of the map is the source node and the value is a list with all the destination points that share a common start.

From that we get that all the destination points are defined by their start. About the destination nodes with multiple sources, we made the decision to allow our delivery agent to visit a destination as soon as even one of their sources is visited. This would provide better space and time efficiency in the real world.

So the concept for this implementation is that we first look among our starting points and, starting with the agent's location as our first node, we calculate all the distances. We pick the node which has the least distance from current and we add it to our tour. Having checked it is a start Node, we have to search the sorted map of destinations and get the list with all the destination points that are starting from the chosen starting point and add it to our list of available nodes. We repeat the same procedure, having as current node the one we chose last, until all the nodes are added in the tour. It is encouraged to consult the pseudo-code, in Appendix A.3, to get a deeper understanding of the described process.

Additionally, an indicative example of the results of the execution at this point of the development is the shown in Figure2.7 where the green markers are our starting points and the red markers are all the destination points. The labels inside the markers indicate the order of traversal.

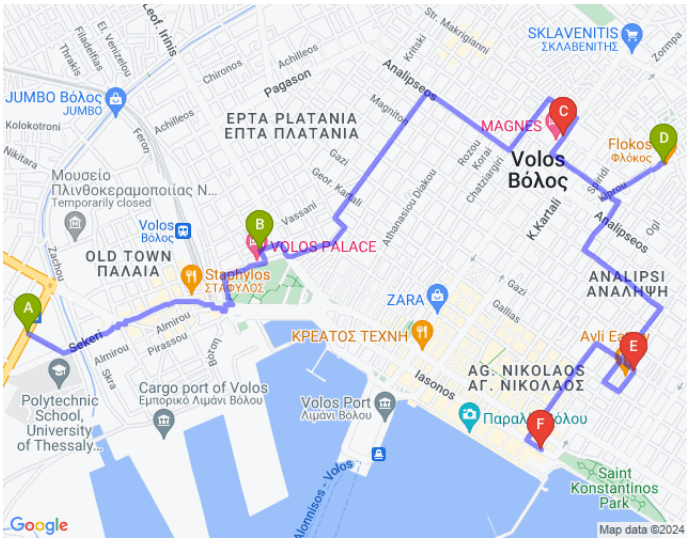


Figure 2.7: Execution Results of Phase 3

2.5 Phase 4: Task Distribution to delivery agents

After completing the problem of searching for the optimal path between two points and finding an efficient way to decide on the order of traversal, we extended a bit more the problem. The new question was how could we manage to deliver the existing packages if more than one delivery agents where available.

In reality, this is the task distribution problem where we have available agents and a flow of packages we want to assign to each one of them. It is a complex and multi-factorial problem, as we have to take into consideration the load-balancing conditions, keeping the tasks fairly distributed, without exceeding the total capacity each agent can carry at any time. Also, in the real world, we must consider some additional factors as the whether forecast and traffic conditions, that may obstruct the normal flow of the distribution, and the time that will be needed to deliver every order. Additionally, in the real world there are cases where a first come- first serve policy must be applied.

So, we made some conventions in order to define our case study. We assumed that our delivery agents start from different spots around the city and there is a maximum capacity that is defined every time by the user. We also considered that only one package arrives at the time. For that, the order of arrival is very important, as the load balancing factor cannot ever be exceeded. The last two compromises we did, were the actual traffic data, which could not been extracted every moment, as it would make the calculations a lot more complex and time costly and we were not strict about the time of the delivery, so there was not an imposed first come-

first serve policy.

2.5.1 Task Distribution Algorithm

The idea behind the algorithm we developed was to use a K-D Tree to store all the locations of interest, which are the source and the destination points of the packages but also include the starting positions of the agents, and every time a new package arrived, we were looking in the Tree for the node that was closest to the starting point of the current package.

After finding the closest package, we found the delivery agent who was responsible for it and then checked if all the requirements were satisfied. The requirements included the load balancing factor and the maximum capacity. The load balancing factor is defined as the total packages assigned divided by the number of available agents. The maximum capacity is the amount of packages each agent is allowed to carry and is defined by the user. If any of the requirements was not satisfied, we created a sub-Tree of the original without the packages of the current agent and searched again for the closest package.

The pseudo-code for the procedure we just described is in Appendix A.4 and taking a closer look of the detailed steps followed, will support the better understanding of the described procedures.

Indicative examples of the execution in this phase can be found in Chapter 4, Fig. 4.7.

Chapter 3

Web Application

3.1 Introduction

In this Chapter we will present the Web Application of our project, which provides a user interface that helps the user interact with the system. We will present in depth all the technologies and tools we used, starting with React.js, a flexible JavaScript library that is ideal for building the user interface, or the front end, of the web applications. We will analyze the fundamental elements and the basic structure of React.js and we will give an insight to the advantages and the numerous possibilities it offers.

The second part of this project will be dedicated to the server-side, or the Backend, of our web Application which was build using Spring Boot. We will explore how we developed our REST API, how we achieved communication with the user interface and how we integrated the algorithmic part in the server side, in order to execute our program and return the results to the user.

3.2 Frontend Technology: React

In this subsection we will explore React, which is a JavaScript library, we will explain all the benefits it offers in general and particularly in our Web Application, we will describe the core concepts of React, we will give an extensive explanation of our Project structure and all the components and in the end we will focus on the integration with server-side.

3.2.1 Introduction to React and Create React App

React [23] is a well known JavaScript[24] library that is highly appreciated among web developers for many years. It was first released by Facebook, currently Meta[25], in 2011 and from 2013 and since it is open-sourced and it is one of the most successful libraries since according to a survey of Statista [26], 40.6% of developers used it in their projects in 2023.

It is very important to highlight that React is indeed a library, and not a framework, as most of people believe. In reality, a lot of developers use those two terms interchangeably, but this is not what really happens. React is not a framework since it is not a ready-to-use option. The existing framework that is also very popular and is related to React is the Create React App [27], which we used in our implementation.

Create React App is an easy to install framework that builds all the basic project structure, which will be further presented in the following chapters, and makes all the starting and setting up process very easy for the developer. It sets up all the environment to make the coding experience very straightforward and easy, especially for a new developer.

3.2.2 Advantages and Disadvantages of React

Already from the introduction we have mentioned some of the advantages of React, but now we will delve a bit more and explain them. We will also examine the disadvantages of React, since no technology or tool is suitable for every project and it is important to show that we have to find the best fit for every situation.

The most important characteristic of React is that it is Component-based, meaning we can build smaller modules, or components as they are called, that are reusable. This is time efficient, we do not need to repeat lines of code and of course it is easier to maintain in the future. The next crucial benefit is the variety of tools and libraries that can be easily integrated in the code. It is one of the richest ecosystems, compared to similar technologies, making the development a lot faster and more efficient. Another advantage we could not fail to mention is the DevTools offered, allowing to see the code changes the moment they are changes without having to restart the server or even refresh the web page. This is an additional comfort for the developer, as it saves plenty of time, especially in the early stages of developing.

Another important characteristic that makes React Applications faster is the virtual-DOM. The conventional Document Object Model (DOM) is the standard way to achieve communica-

tion between the programming languages and the web pages, all the page elements are saves as objects and nodes. The virtual-DOM provided by React, implements a virtual copy, that stops redundant and unintentional renders of the page components. This makes the interaction with the web page way faster, enhancing the overall performance. Additionally, using the virtual-DOM achieves better Search Engine Optimization (SEO), getting better rankings and showing the React-based application at the top of searches, which is an important feature in the business world.

Some more advantages are the support offered by Facebook, making it a reliable to build modern applications that will not be outdated soon. Except for Facebook, there is also a big active community of React, which makes it easier to find support by people how share the same problems. Lastly, it provides cross-platform functionality, that means that it supports web and mobile applications, it also allows scalability, which is why plenty of the most-used applications are built in React, and is compatible with all of the browsers, giving consistently sufficient user experience and reliability to the applications.

Of course, we could not claimed that there are just advantages in using React, without mention any negative aspects. Firstly, as it is so widely used and open-sourced, it is updating very often, creating the need to stay up to date with all the new changes. Another tricky characteristic of React is the language: JSX language is a mixture of HTML and JavaScript and many new developers consider it to be a bit challenging at first. However, even though the required adjustment time, it is the only way to build a component based language, that provides improved readability. The last downside we will focus on is the difficulty in integrating a new project with existing ones, especially if they were not initially built in React. It is sometimes hard to refactor all the components and keep the architecture correct, and the difficulty rises if a many updates have elapsed between the two projects.

3.2.3 Core Concepts of React

To get a better understanding on how React works and how it is different than relevant libraries, we will explore the core concepts of it: JSX language, components, state and props and hooks.

Starting with JSX, which stands for JavaScript JavaScript XML, is the specific language used in React, that combines the HTML elements with JavaScript language. In reality, we keep most of the HTML form and add inline JavaScript elements, closed in curly brackets. Addition-

ally we can use JavaScript structures like methods that are instantiated in specific lines of the code.

Moving on to the next core element of React, the Components. Components are the building blocks of this module based architecture and is in reality the element that allows re-usability inside the code. The most contemporary components are called Functional Components and include a function definition that includes all the page that will be rendered inside a return statement. The key for the re-usability is that we can import all of the component into another component, simply by instantiate it in one line. This is also very helpful practice for code maintenance.

The next core element of programming in React are States. States are objects that keep useful information about the data, we could abstractly think of them as variables, but they store a collection of information about our objects. Every time the state of an object changes, then that object is rendered again. The change of state may be caused by user interaction on the page, a click maybe or even a respond from the Backend. The states are holding the information in the interior of the component. However we can have local or global states, where the local states are changed and used only on the inside of a method, while the global ones can be accessed and changed by all the methods inside the component.

As we got familiar with the possibilities inside one component, we shall now explain what happens when a component is instantiated inside another component. The parent component that inherits the child component, does not have access to either the local or the global states. The way the two components are sharing information is through props. Props are read-only elements that are passing to the parent component from inherited one.

The last basic concept of React is Hooks. These features are comparatively new additions to React and are build-in features that are responsible for different functionalities. We already discussed about state and the way to use a state is through the useState() hook that instantiates it. There are many other hooks, such as useEffect(), useRef() and useCallback(). Every one of them performs a different operation, allowing interaction with the user, getting responses from the server or improving the efficiency of the code, for example through avoiding the unnecessary renders.

Figure 3.1: Set up File System of React App

3.2.4 Project Structure and Components

Having established all the advantages of using React in general and what are the general information around it, we shall now proceed to how we structured our project and what components we used to perform our User Interface.

We used React App as our React Framework, which means that after the set up, all the file system was built automatically for our project and the results for the standard set up are shown in Figure3.1.

Inside the src File we observe the App.js file which is the main file that will connect with our custom components. Additionally it is worth mentioning the package.json and the package-lock.json files that are keeping all the dependencies our Web Application needs and are vital for starting and launching our application on the web.

Inside the src folder we were able to create all our custom components which we will now describe thoroughly:

1. Main Component: it is where the virtual-DOM is established and all the other components are called. The purpose of this component is the coordination of all the others.

2. Header Component: a simple header component that redirects to the first page

3. Footer Component: a simple footer component with links for the Github repository

4. Map Interaction Component: a component that is responsible for user interaction with the map. It provides a Google Map with an interactive search box, where the user may search for a location and by doing that a marker is added in that location and the coordinates

are saved and sent to the server. This component is inherited by two other components, providing individual use in each case.

In this component we were extremely benefited by the easy interaction provided by React in order to work with Google Maps, only by using a unique API Key.

5. Agent Location Component: this component is handling all the agent information we need to collect and sent to the server side. It inherits the Map Interaction Component and uses it to collect the coordinates by the locations entered by the user. Additionally there is a number input field that the user can insert the maximum capacity each agent can carry. The number input field, as some other properties used in React, where added using React Bootstrap [28], a library with useful components that are ready to plug-in in any React application.

   After gathering all these information, this component is responsible to send them to the server. We will give more information on that in the following section.

6. Package Location Component: this component has a similar structure as the agent location component. This component also inherits the Map Interaction Component and is responsible to gather all the data about the source and destination coordinates of all the packages. After selecting all the necessary information, this component also communicates with the server side and sends this information in order to do the processing.

7. Map Display Component: this is the last component we constructed and includes a slideshow responsible to display all the different maps with the routing and the path every agent has to follow. It is also a component that connects with the server side, since those maps are the result of the algorithmic execution and a list with the desired URLs is returned to the user.

3.2.5 Integrating with Spring Boot

In this final section of React, we will present how we established communication with the server side, our Spring Boot Server. React and Spring Boot is a very popular combination for Web Applications, as both are very flexible and modern, and always expand their ecosystems, adding new features and giving the developers more possibilities. This makes them extremely agile and appealing to use and this is why we chose them.

We will now proceed on explaining how these two are communicating. As in every Web Application, we create end-points on which the server side and the client side will connect and exchange data. On our client server the way we decided to create our end points is the HTTP Fetch [29].

The benefits of using HTTP fetch, is that HTTP fetched is a native built-in JavaScript API, ready to use in all modern browsers, with no need of extra setting up. It handles basic server requests and is more simple to use compared to other technologies. Additionally, it is indicated when there are not complicated requests to handle, since it is more simple, light-weight, and thus contributing to better performance and faster serving of the requests. A downsize is the need to handle errors manually, as the system provides limited error report.

There are two types of methods that are supported by fetch and are commonly used when communicating with the server side: the POST and the GET requests. The former method is used when the client side wants to send information to the server side, usually in an enclosed form, for example a JSON[30]. The server side must have the according end points to read all the data sent. The second method is used to obtain information, or data, from a specified source. For instance, it could be data sent from the server-side or from a database and the client side must adjust the according endpoints to store and access those data. There are the two universal ways to exchange data from the server side to the client side, and the other way around.

We also need to mention that React and Spring Boot are by default assigned to different ports of the localhost. The default port for React is 3000, while for Spring Boot we have 8080, which also need to be specified, so we know where we are sending the data and from where we expect to receive them.

The way we perform our POST Methods is shown in Appendix B.1, and is by specifying the location and the exact directory our data will be sent. Additionally, all our data are sent inside a body that is in JSON format so they can be received from server. We have to add some code to get the server responses and to handle errors that may occur.

On the other hand to perform a GET Method, we have to define again the exact location we are expecting to get the results from but we also have to create the appropriate states (useState() hook) to save the received data and be able to handle them. Additionally, we may use useEffect() hook that is responsible to execute and render the enclosed code only when the expected data is mounted, and not rendering repeatedly when there is nothing sent yet. This supports the better performance of the application.

This is all we have to do on the client side to send and receive data. The server side has to create their own end-points, as it will be explained analytically in the following section.

3.3 Backend Technology: Spring Boot

This subsection is dedicated to Spring Boot, a framework of Spring. We will give a general introduction and we will list some of the advantages of this widely used framework, without omitting the downsides. We will describe how we used it in our implementation, emphasizing in the communication with client side, React.

3.3.1 Introduction to Spring and Spring Boot

Spring[31] is a Java-based and open-source framework that was built to aid developers implement applications at first, and, with some extensions, web applications. It is still widely used, especially in when the desired result is to build an application with plain Java objects and do not depend on any framework and constant changes of environments and releases.

However, in 2014 Spring Boot[32] was released by Pivotal Software, currently part of VMware[33], and the idea was to give developers a framework suitable for building applications with minimum effort. It was free of many prerequisites and it provide a simplified mechanism where the application could just be executed.

All the above, make it the prevailing framework for all the Java developers around the world and it keeps evolving and following the state-of-the-art practices. It is also claimed that even if it ever becomes obsolete, the core concepts developed in Spring Boot, will be adopted by the descendant frameworks.

3.3.2 Advantages and Disadvantages of Spring Boot

As we made clear the importance of Spring Boot Framework, we will now specify the reasons it is so highly adopted by developers. Of course, we will not hide the drawbacks, as it is important to choose the technology that suits best our project goals each time.

The biggest advantage Spring Boot offers is that includes a variety of things needed to run an application, saving a lot of time in the beginning of a project. For instance there is no need for utilizing WAR files, which are similar to the common JAR files[34] but are responsible for

the web applications. There is also no need for XML Configuration, compared to Spring, as it happens automatically. The last factor of major importance is the embedded servers, vanishing the need for establishing an external server and connecting the application with it. All the above are structuring a framework that assists the smooth beginning of building an application and the easiest of deployments, as the application is simply executed.

The success of Spring Boot is also based on the fact that it still belongs in the ecosystem of Spring. It has adopted plenty of the best elements of Spring, such the ability to build stand-alone applications, that are robust, considerably light-weight and reliable through time. As it belongs in the Spring family and is so widely used by the developers around the world, it maintains a very active community around it, making it easier to solve individual problems.

However, there is no technology that is perfect and can work for every project and Spring Boot is no exception. For many people Spring Boot is consider hard to learn in the beginning of the programming journey. It is considered complex to learn all the mechanics of this framework, in order to use it at its full potential. Additionally, all the automation in the configuration, burdens the system with a resource overhead, as there are files and dependencies that may no be used, but they are not to be deleted. So, if the goal is a strict and constrained environment, in terms of resources, Spring Boot might not be the ideal choice.

3.3.3 Spring Initializr and Maven

Having mentioned the automation in configuration and all the easy starting process, we shall now delve a bit more to that. Spring Boot offers an online tool named Spring Initializr [35] that the user can deploy to initiate a Spring Boot Project in very few steps.

The user defines whether the project will be in Java, in Kotlin[36] or in Groovy[37] languages. Additionally there is the option for Maven or Gradle, considering the tool that will be used to build the project. There are some more information to be specified, as for the project name and description, the type of package that will be generated (JAR or WAR) and the version of Java. Some dependencies, such as web tools, can be specified there and are automatically added in the configuration files of our project. After making our choice for all the above, we are ready to generate our project and let Spring Boot do the automatic configuration for us.

We chose Java and Maven for our project, as it is the most widely used combination. But, let's give some further information on Maven and its role in our system. Maven performs the task a Makefile would do in other languages, like C/C++, which means that it is responsible to build

our project. It holds all the useful information about dependencies, plugins and correct versions of our tools. All this data is stored in a Project Object Model XML file, the well known pom.xml file, that is vital for the execution of our project. This file is also automatically created during the configuration process, but can be modified manually if we want to add some additional dependencies.

3.3.4 Application and Controller

After completing the configuration process our application is set up and we can start our development. In the configuration process, a file with our project name has been generated having the annotation:

```
@SpringBootApplication
```

This annotation is enclosing three annotations that allow all the automatic processes.

```
@Configuration, @EnableAutoConfiguration, @ComponentScan
```

This generated file is, and is suggested to remain, very short in length and all its purpose is to Run our Application. It works as the main class of our project and this is the java file we need to run in order to begin our web server and all the Application.

The code needed, in order to have a functional *Main* class that initiates our Web Application is presented in Appendix B.2. We observe there some necessary Spring Boot imports, but also a command starting with ”package”. This command must be included as header in all the java the classes we use in our Application and it is the way Spring Initializr establishes communication inside the Java Environment.

The next important java class we have to implement in our project is the Controller Classes. In our simple and light-weight project one controller was enough to handle all our needs, but generally more than one controllers can be supported by Spring Boot.

The Controller class is responsible for connecting the client side and the server side and it is where the well known API is built. There we define all the functions that are responsible to respond to the POST and GET requests of the client side and then execute the according classes to make the necessary computations.

Our Controller class contains three basic annotations:

```
@RestController, @CrossOrigin(origins = "*"), @RequestMapping
```

The first annotation, Rest Controller, defines that this class is the responsible for our REST API, and includes the annotation about Response Body, which is the confirmation we sent to the client after receiving successfully or failing to receive the data.

The next annotation is Cross Origin annotation, which is part of the Cross-Origin Resource Sharing (CORS) mechanism, that is giving permission to communicate with sources from different domains or ports. In our case, it is the latter we want to permit, since React and Spring Boot are using different ports of local host. This is how we give permission to React to share data in localhost:8080 (Spring port) and not get blocked by HTTP.

The last annotation is Request Mapping, which is asking mapping with the requests sent to HTTP. We can specify a particular origin, for example a specific page of the Web Application, or a specific method of mapping, that is to say POST or GET methods. In our case, we used the plain annotation and made specific requests in the included methods.

This lead us to the next topic, which is what is included inside a Controller. The Controller must contain very specific end points that are ready to receive the data in the form the client side is sending them, otherwise it is impossible to get the fetched data. We implemented three functions to connect with the end points of React:

1. Function for the agent information: this method is annotated with a Post Mapping annotation and the according page where we want to include in our scope. It is where the client is also sharing the data.

   There we have used an instance of a custom class named ArgsHolder that has as attributes all the data included in the JSON sent by React. By doing that, we are ready to parse the JSON elements in different variables and use them as needed. In Appendix B.3 there is the back-bone of this implementation to provide visual representation of the above description.

2. Function for the package information: this method is very similar with the agent information method. It also requires a Post Mapping Annotation, but of course we change the origin. In this case, we are only getting a list of coordinates, as String, so there is no necessity for extra argument class here and the data is received as in the original form. In both our Post Mapping methods, we include the Request Body annotation that protects the user from sending an empty body of data, or skip some of the required information. If the body is empty, the HTTP request is not completed successfully

3. Function for the resulting maps: the last method of our controller is controlling a GET method request, thus it requires a GetMapping annotation in the according page. In this case, after our program has been executed and the expected results have been produced, in our case the Google Maps images, we are ready to sent them to the client. In Appendix B.4, there is the GET method we need to return our results.

The last thing to mention is that our Backend algorithmic part was integrated in the current project. The prior *Main* Class was refactored as *AlgorithmRunner* Class, that is executed by the Controller as any traditional method, after all the required arguments are sent from the client.

Chapter 4

Navigation in the Application

4.1 Introduction

In the present Chapter, we will present a comprehensive high-level description and navigation of our web application. We aim to give an overview of the structured followed, explain the purpose of every page and the main elements inside it and all of that will be presented keeping in mind the User Experience.

The chapter is articulated in chapters for each one of the pages: starting with the delivery agents, then the packages and finally the obtained maps. To get a deeper understanding and to visualize all the elements described screenshots of each page, will be provided in each step.

4.2 Delivery Agents Page

The first page we are going to dive into is the page that is responsible for handling agents and all the necessary information about them.

As shown in Figure 4.1, this is what the user encounters as enter the web application. We notice that there is a very simplistic design in the page including only the necessary elements needed, in order to provide the user interaction with the system.

On the left side, there is a large interactive map where the user can search different locations, zoom in and out, and click on specific locations to see information about them, as they are registered in Google Maps.

To begin interacting with our application, the user can add the initial delivery agent locations by clicking directly into the according point of the map or by typing the address into the Search

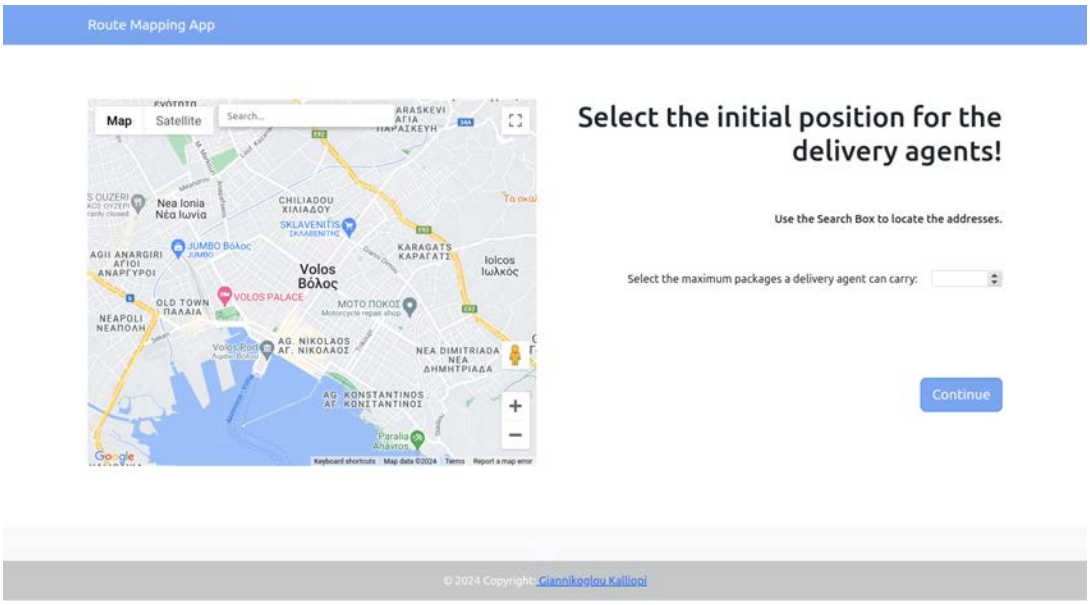


Figure 4.1: Display of initial delivery agents page

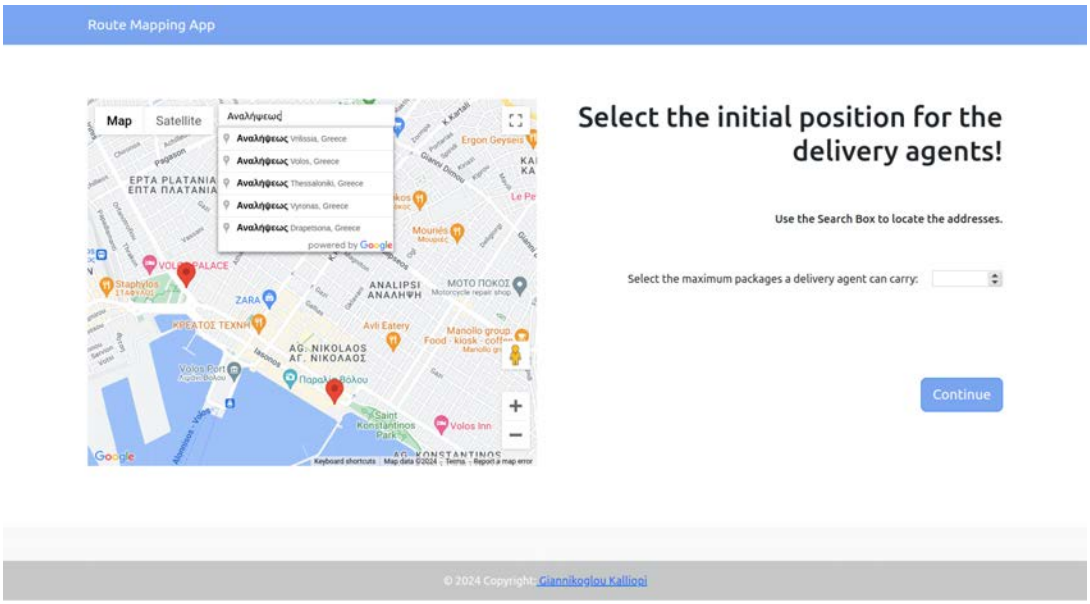


Figure 4.2: Selecting delivery agents locations

Box. The user has the option of searching a location with either the official address or with the place names, as registered in Google, such as a university, a store or a restaurant. Of course, instant coordinates are also recognised, similarly to Google Maps. As illustrated in Figure 4.2, there is an activated auto-complete feature while typing, making the user experience easier. To insert the address, the user may either press the 'Enter' key, or click the location on the auto-complete menu.

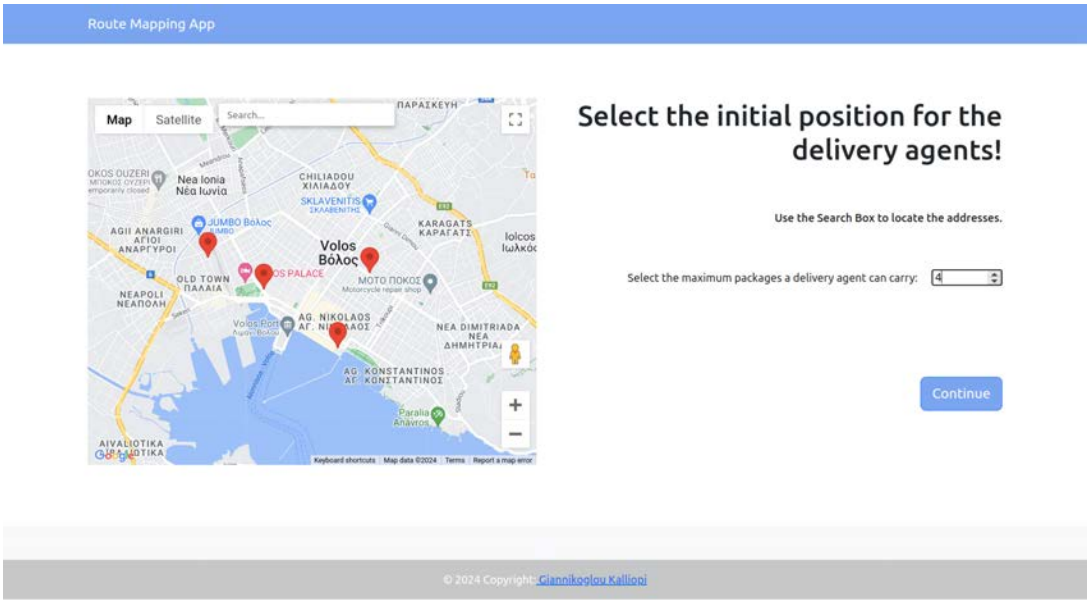


Figure 4.3: Selecting maximum packages

Every time a location is registered successfully, a red marker is appearing in the map. So when we have inserted all the delivery agents locations, they will appear with red markers, as depicted in Figure 4.3.

Once we have finished with the locations, we can adjust the maximum packages each delivery agent can carry during their ride. This parameter is specified via the numeric input form located in the right part of the page. The user may type the number, or use the existing arrows. If this part is skipped, the system automatically assigns a minimum capacity of 1 package per delivery agent.

For our example, we inserted 4 agents in the places marked in red as depicted in Figure 4.3 and we defined a maximum capacity of 4 packages.

When all the locations are inserted and the desired maximum capacity is defined, the user may press the 'Continue' button and move on to the next page.

4.3 Packages Page

The next page we are going to present is the Packages page. As illustrated in Figure 4.4, it has a similar structure to the delivery agents location page, and is responsible to gather all the source and destination locations of the desired packages.

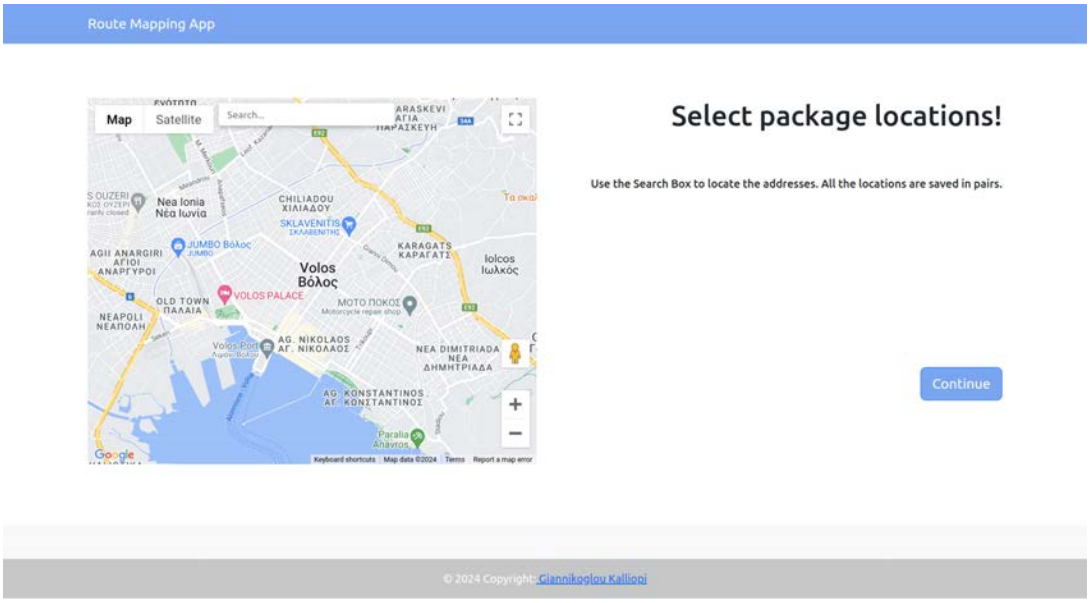


Figure 4.4: Display of packages page

The page includes another interactive map, with all the functionalities activated, such as the auto-completing and zoom in and out. It is a cleared map and the initial locations of the delivery agents are not displayed, to avoid confusion with the package locations, in case some of them are located in the same points as the agents locations.

The user follows the same procedure as in the previous page, and enters all the locations one after the other, by clicking or by typing the address. Each time a location is inserted successfully, a red marker appears. It is worth mentioning that all packages must be added in pairs and the pairs cannot be changed afterwards. Additionally, if the locations do not have an even number, meaning one of them is not having a pair, the system by default removes the last added element.

Our example package locations are illustrated in Figure 4.5. We have chosen points throughout all the city and avoided remote sites, in order to show the average usage example. In our example, we also chose two locations as shared sources for more than one destinations, thus we expect some of the routes to have more destination points that source points.

By pressing the blue button ’Continue’, the user is directed to the next and final page of the application, the Map Display page.

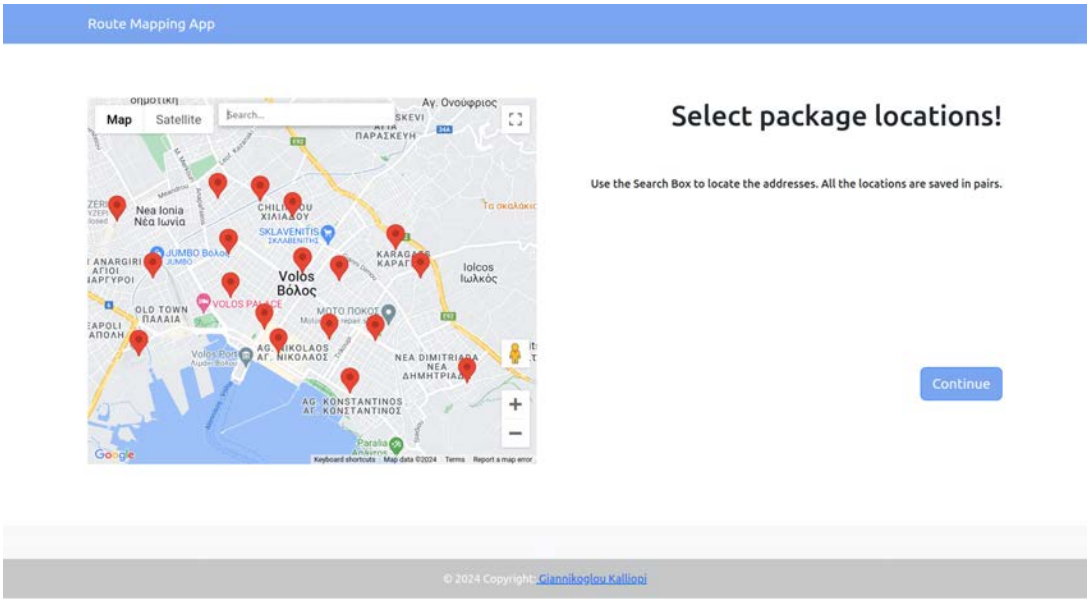


Figure 4.5: Display of the inserted packages

4.4 Final Maps Page

The last page of our implementation is the result page, where the final static maps are shown. As shown in Figure 4.6, the page has a reactive slide show, where the user can use the arrows (left and right) or click of the preview on the bottom left of the page and access all the images.

We have one map image per agent, showing the route each of them has to follow. The results of our example are shown in Figure 4.7. All the routes begin from the blue markers, that indicate the starting positions of the delivery agents. Then the path is marked in blue and the source points are depicted with green markers, while the destination points are presented in red. The labels are in alphabetical order to support the understanding process and help the user comprehend easier the correct order of the mapping.

It is worth mentioning that both the first 4.7α′ and the second 4.7β′ sub-figures have more destination points than the sources. The reason for that is the shared sources we used in our example. More specifically, in the second sub-figure both the destinations B and C have as common start the location A and the same happens for destination points D and E in the first sub-figure.

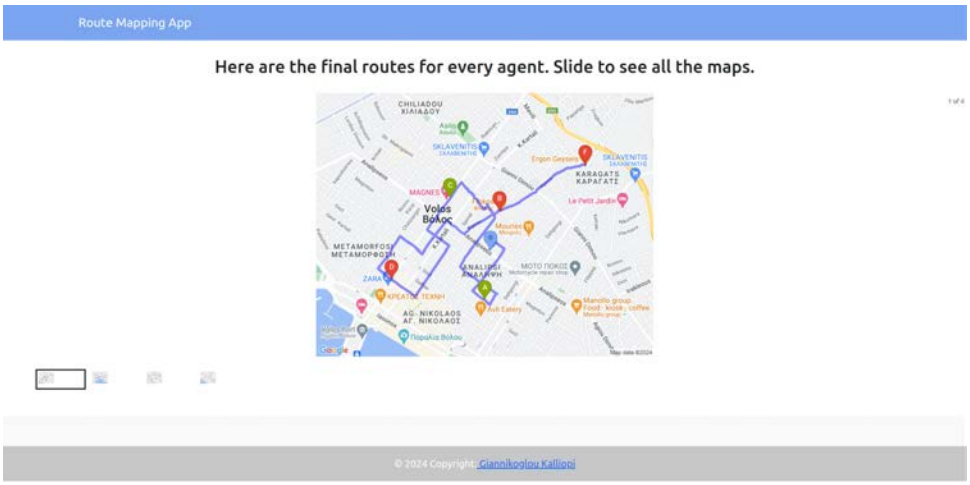
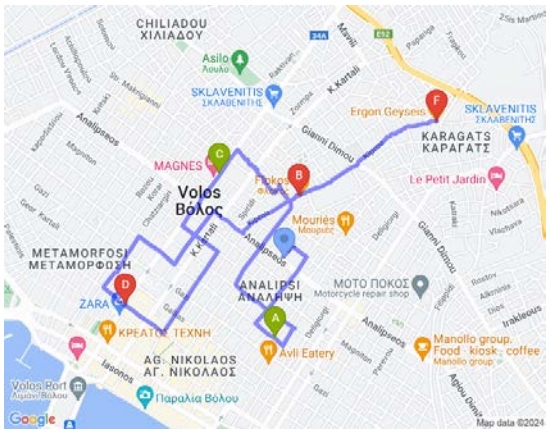
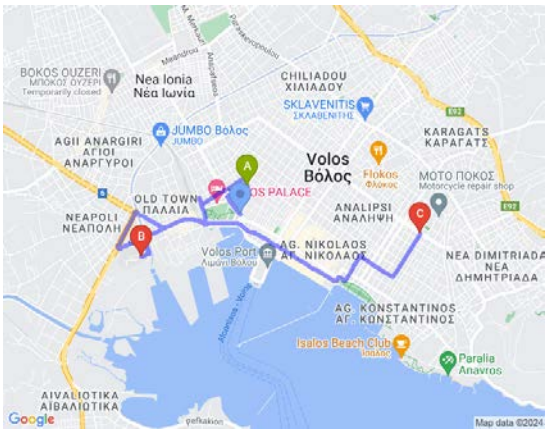


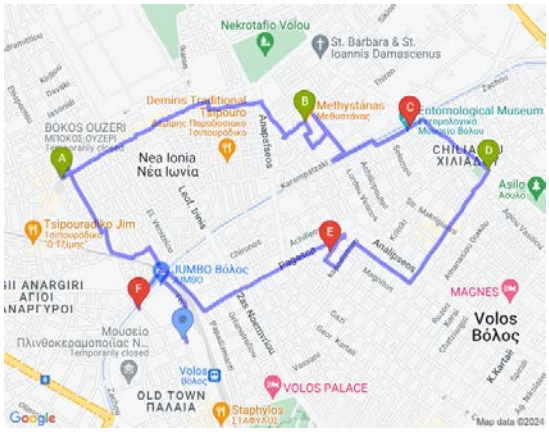
Figure 4.6: Display of the routes for all the agents



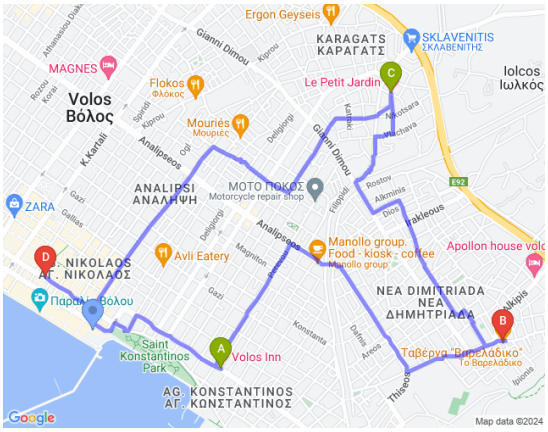
Result static map 1



Result static map 2



Result static map 3



Result static map 4

Figure 4.7: Overall results of the 4 maps

Chapter 5

Conclusion

This final chapter concludes the present thesis, in which has explored the complex problem of Route Mapping within a city, considering we have multiple delivery agents which are assigned with packages. Additionally, a minimal and lightweight web application was implemented to provide a user interface, visualizing the results of our algorithms.

In this chapter, we will summarize our study, discuss the results achieved with a critical view and propose some directions for future work.

5.1 Summary

The main purpose of this study was to explore the multi-factorial issues of route mapping and task distribution. For the route mapping part, we implemented the A* search algorithm to find the optimal path between two points, which was the basis we used in all the following steps.

The next algorithm we used was Nearest Neighbor Heuristic, which was used with a small alteration of specific start, instead of the random beginning it normally has. This algorithm was used to generate an order of traversal for one source and multiple destination points.

Our next implementation, was a custom algorithm that was based on the philosophy of Nearest Neighbor Heuristic search, and was used to solve the problem of optimal traversal order, considering that all the points are in pairs, source and destination. This added some limitations to our system, as not all the points where available to be visited from the beginning.

The last problem we attempted to solve, was the task distribution problem, which is a very complex problem and is a hard timescheduling problem in real life. We proposed an approach on how the tasks (or the packages, in our case) should be assigned to the available agents,

prioritizing some factors and making some compromises on others. The implemented algorithm was not based on any specific algorithm but it followed the fundamental logic behind this genre of algorithms.

The algorithmic process was supported by the implementation of a web application. The application has a simple, yet functional, structure that was responsible to provide a user interface to project our algorithmic findings. It was also used to check the scalability of our project and the performance in terms of execution time, as the input data was growing.

5.2 Results

The results of our study were quite satisfying; however, it is beneficial to maintain a critical perspective when evaluating any work and thus, this is what we will discuss in this section.

Starting from the end of the algorithmic process, the task distribution problem, we can safely claim that the given results are reasonable. The task distribution problem, as we have established it in our case study, is highly sensitive in the order the packages are entering the system. As only one of them is visible by the system at the time, the load factor of every agent at the present moment highly affects the assignment process. It is an algorithm that follows the fundamental ideas of the task distribution problems, that uses fast and effective searching mechanisms, using KD-Trees, and is flexible in adjustments. For all the above, it is proven that every potential client could use the existing implementation, by making micro-adjustments to cover their individual needs.

The next major component of the algorithmic process, is the algorithms used to find the optimal order of traversal, the Nearest Neighbor Heuristic and our alteration on that. We consider the heuristic approaches the most suitable idea for that matter, as they are solving the Travelling Salesman Problem. It is an NP-Hard problem that could not be solved in a polynomial time, which is not affordable for any system. We chose a greedy approach, that makes a choice only influenced by the present position and does not predict anything for the future. Although it is a naive approach, it is the fastest approach we could have. It converges in the minimum number of iterations, compared to other greedy approaches, and in the worst case it gives the average solution, compared to the optimal. Additionally, the simplicity of the implementation is what made it so flexible, allowing us to make our alterations to cover our needs, in the third phase of the implementation.

We consider that other heuristic approaches could also be used here. A more complex solution could achieve results closer to the optimal, having the trade off of the execution time. We should also keep in mind that more complex implementations, do not always have the ability to adjust in our specific needs.

The last part of the algorithmic process was the implementation of A* search to find the optimal path between two points. We used the existing algorithm, with no alterations or adjustments. It is a very ”smart” algorithm, that makes decisions considering both the past steps and heuristically calculates the future steps. It is based on the widely used Dijkstra algorithm and it is considered to be the absolute path-finding approach for general purposes, when no special treatment is required.

The last thing we have to evaluate is the web application implemented, providing the user interface. It was a very straightforward implementation with no ornamentation, that was build solely to connect the user with our algorithm. For that it is pointless to judge it as a complete web application and examine factors, such as the security. The implementation was done using modern and competitive technologies, such as React and Spring Boot, that are widely used from web developers globally. It served the educational process we were expecting, and we could claim that our needs were covered to the maximum. It provides a user friendly, easy to comprehend environment, where the user may interact with our algorithm and have access to the produced results.

5.3 Future Work

While our study proposed a solution to the complex issue of route mapping within a city, we could not consider that the topic is exhaustively explored. It is a large field for further studies, and many of the existing proposals, including ours, have plenty of room for optimization.

Building on our proposed solution, we propose the following as future work:

1. There is room for additional experimentation with algorithms that solve the optimal order of traversal problem. They could be applied in our system and compare the results in terms of time and efficiency.

2. The web application could be improved, by developing of a robust application that could be used in the real world, allowing users greater control over the parameters, especially in the task distribution algorithm

3. Variations of our case studies could be explored and integrated in out system by making the appropriate changes and adding features respectively

4. Integration of real-time data in the existing solution, for instance the traffic or the weather could make our implementation more accurate and reliable for real world usage

5. Allowing simultaneous routing for different cities, would make our system more agile. Now, we are focusing on a single city, but adding bigger maps to our system, could pave the way for large-scale route mapping, for example professional trucks.

Bibliography

[1] Circuit route planner. https://getcircuit.com/route-planner.

[2] Maposcope- multi-stop route planner. https://maposcope.com/.

[3] Google maps api, google for developers. https://developers.google.com/maps.

[4] Oracle java official page. https://www.java.com/en/.

[5] Intellij idea jetbrains official page. https://www.jetbrains.com/idea/.

[6] Github official page. https://github.com/.

[7] Github repository for routemappingproject. https://github.com/KalliGiannikoglou/RouteMappingProject.

[8] Github repository for the backend of routemappingproject. https://github.com/KalliGiannikoglou/Backend-of-RouteMappingProject-SpringBoot.

[9] Github repository for the frontend of routemappingproject. https://github.com/KalliGiannikoglou/Frontend-of-Route-Mapping-Project-React.

[10] Visual paradigm, uml online tool. https://www.visual-paradigm.com/.

[11] Openstreetmap. https://www.openstreetmap.org/.

[12] Jsoup java xml parser. https://jsoup.org/.

[13] Java interface map. https://docs.oracle.com/javase/8/docs/api/java/util/Map.html.

[14] J. Wetherell. java-algorithms-implementation, github repository, 2017. https://github.com/phishman3579/java-algorithms-implementation/blob/master/src/com/jwetherell/algorithms/data_structures/KdTree.java.

[15] National institute of standards and technology- manhattan distance. https://xlinux.nist.gov/dads/HTML/manhattanDistance.html.

[16] National institute of standards and technology- euclidean distance. https://xlinux.nist.gov/dads/HTML/euclidndstnc.html.

[17] Gilbert Laporte. The traveling salesman problem: An overview of exact and approximate algorithms. *European Journal of Operational Research*, 59(2):231–247, 1992.

[18] A Hanif Halim and IJAoCMiE Ismail. Combinatorial optimization: comparison of heuristic algorithms in travelling salesman problem. *Archives of Computational Methods in Engineering*, 26:367–380, 2019.

[19] Daniel J Rosenkrantz, Richard E Stearns, and Philip M Lewis, II. An analysis of several heuristics for the traveling salesman problem. *SIAM journal on computing*, 6(3):563–581, 1977.

[20] Haider A Abdulkarim and Ibrahim F Alshammari. Comparison of algorithms for solving traveling salesman problem. *International Journal of Engineering and Advanced Technology*, 4(6):76–79, 2015.

[21] Java interface list. https://docs.oracle.com/javase%2F8%2Fdocs%2Fapi%2F%2F/java/util/List.html.

[22] Java interface sorted map. https://docs.oracle.com/javase/8/docs/api/java/util/SortedMap.html.

[23] React official page. https://react.dev/.

[24] Javascript official page. https://www.javascript.com/.

[25] Meta homepage (prior facebook). https://about.meta.com/.

[26] Statista survey: Most used web frameworks among developers worldwide, as of 2023. https://www.statista.com/statistics/1124699/worldwide-developer-survey-most-used-frameworks-web/.

[27] Create react app page. https://create-react-app.dev/.

[28] React bootstrap. https://react-bootstrap.netlify.app/.

[29] Javascript info for fetch. https://javascript.info/fetch.

[30] Json format. https://www.json.org/json-en.html.

[31] Spring official page. https://spring.io/.

[32] Spring boot page. https://spring.io/projects/spring-boot.

[33] Vmware official page. https://www.vmware.com/.

[34] Jar file overview by oracle. https://docs.oracle.com/javase/8/docs/technotes/guides/jar/jarGuide.html.

[35] Spring initializr. https://start.spring.io/.

[36] Kotlin official page. https://kotlinlang.org/.

[37] Groovy official page. https://www.groovy-lang.org/.

APPENDICES

Appendix A

Algorithms

In the present Appendix A, we will present the pseudo codes for all the Algorithms implemented in the algorithmic part of our implementation and are extensively analysed in Chapter 2. They are presented to assist the comprehension of the reader, giving a detailed and concise view of the process we followed. It is important to mention that pseudo code is a combination of programming and natural language, providing a higher-level of understanding, and it cannot be executed in Java programming language in that form. For the original source code, the reader is encouraged to visit the Github page of the repository[7].

A.1 A* Search

The important algorithm implemented in Phase 1 was the A* Search (A star search). This is what we will present below, in order to give a more detailed representation of the steps we followed in our development process.

Algorithm 1: A* Search

```
1  Open ← add start node;
2  Close ← empty set;
3  start.gVal ← 0;
4  start.hVal ← Dist(start, end);
5  while Open ≠ ∅ do
6      curr ← Node with min fVal;
7      if curr ≐ end then
8          return constructFinalPath()
9      else
10         for each Neighbor N of curr do
11             if N ∉ Close then
12                 calculate N.gVal, N.hVal;
13                 N.prev ← curr;
14                 if N ∈ Open and N.gVal < Open[N].gVal then
15                     Open ← N;
16                 Close ← N;
```

A.2 Nearest Neighbor Heuristic

In this subsection we are presenting the pseudo code for the implemented method of Nearest Neighbor Heuristic Algorithm that was described thoroughly in Chapter 2. We have added the alteration of specific starting point, which is the location the delivery agent starts and is located in the first position of the nodes array.

Algorithm 2: Nearest Neighbor Heuristic

Data: $nodes, nodes.distances$

```
1  tour ← null;
2  toVisit ← nodes;
3  minNode ← null;
4  curr ← toVisit(0) # the first entry is the agent position;
5  curr.checked ← true ;
6  while toVisit ≠ empty do
7      minDist ← MAX VALUE;
8      for each distanceD ∈ curr.distances do
9          if D ≤ minDist then
10             dChecked ← check if the index of D is unvisited in curr;
11             if !dChecked then
12                 minNode = nodes[d];
13                 minDist = D
14     tour ← curr;
15     remove curr from toVisit;
16     curr ← minNode, minNode.checked = true;
```

A.3 Traversal Order Algorithm

This pseudo code provided is our alteration of the Nearest Neighbor Heuristic algorithm in order to handle multiple source and multiple destination points.

Algorithm 3: Traversal Order

Data: $availNodes(startNodes), tour(null)$

$minNode \leftarrow null$;
$minDist \leftarrow$ MAX VALUE;
if $tour.empty()$ **then**
    $curr \leftarrow availNodes[0]$ # the first entry is the agent position;
    $tour \leftarrow curr$
else
    $curr \leftarrow tour[last]$ # the last chosen element ;
Clear $curr.distances$;
if $curr$ *is Start Node* & $curr$ *Not visited* **then**
    for *each* $dest \in curr$ *list of destinations* **do**
        $availNodes \leftarrow dest$;
    $curr \leftarrow$ mark as visited ;
Remove $curr$ from $availNodes$;
if $availNodes.empty()$ **then**
    $return$;
for *each node*$n \in availNodes$ **do**
    $curr.distances \leftarrow$ calculate Manhattan Dist $curr, n$;
for *each distance*$D \in curr.distances$ **do**
    if $D \leq minDist$ **then**
        $minNode = nodes[d]$;
        $minDist = D$
if $minNode \neq null$ **then**
    $tour \leftarrow minNode$;
    Remove $minNode$ from $availNodes$;

A.4 Task Distribution

The pseudo code for the algorithm we implemented to distribute the packages among the agents. It is presented in order to visualize the theoretic description of the according section.

Algorithm 4: Task Distribution

Data: $src, dest$

```
1  tree ← packages;
2  nearestPkg ← nearestNeighborSearch(1, src);
3  while true do
4      if nearestPkg == null then
5          break;
6      else if nearestPkg == empty then
7          break;
8      else
9          agentId ← assignedPkgs(nearestPkg) ;
10         if agentId ≠ null then
11             for each agent A assigned with this point do
12                 agentsPkgs ← getPackages() ;
13                 loadFactor ← totalPkgs/totalAgents ;
14                 if agentsPkgs < MAX CAPACITY & agentsPkgs ≤ loadFactor
                    then
15                     add(src), add(dest), totalPkgs = totalPkgs + 1;
16                     break ;
17         |
```

Appendix B

Web Application Code

In this Appendix we will give some indicative code considering Chapter 3 of the current thesis. It will not provide many lines of code, neither it will be code ready to execute, but it will serve the purposes of deeper comprehension of the explained procedures. The reader is encouraged to consult the referenced parts, in parallel with the theoretical description in the according chapter.

To see the source code of the whole Web Application, please visit the according Github Repositories, for the Frontend [9] and for the Backend [8].

B.1 HTTP Fetch React

This is a short example of how our POST requests are performed. We define the fetched location, we add the headers and the body, which must send the data to server in JSON format. We also have to handle the responses and provide some error handling process.

```
await fetch('http://localhost:8080/agents', {
    method: 'POST',
        headers: {
            'Accept': 'application/json',
            'Content-Type': 'application/json',
        },
        body: JSON.stringify({"agentsLocations": locations,
        "maxCapacity": maxCapacity})
    })
```

```
    .then((response) => response.json())
    .then((responseJson) => {
        return responseJson;
    })
    .catch((error) => {
        console.error(error);
    });
```

B.2 Spring Boot Application Java Class

Here is the automatically generated Java Class that is the ”brain” of the Application. It contains the main class and it is suggested to be kept short and perform the simple but crucial task of running the Application.

```
package com.RouteMappingProject.RouteMapping;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class RouteMappingApplication {
    public static void main(String[] args) {
        SpringApplication.run(RouteMappingApplication.class, args);
    }
}
```

B.3 Post Mapping Example

This is a plain example of how the data sent from React, are received by the server. We also notice the way our arguments are passed as an integrated object and not as separate arguments.

```
@PostMapping(”/agents”)
    public ResponseEntity getAgentsLocations(
```

```
                    @RequestBody ArgsHolder args){

        allAgents = args.getAgentsLocations();
        totalMaxCapacity = args.getMaxCapacity();
        return new ResponseEntity("Success", HttpStatus.OK);
    }
```

B.4 Get Mapping Example

This is a simple example of how the data is sent from server to client. It is required to have some local variables that store the result of the execution of the algorithmic part, in order to send it to the client.

```
@GetMapping("/maps")
    public List<String> sendMaps() throws IOException{
        return algorithmRunner.maps;
    }
```