ERICSSON

DEPARTMENT OF RESEARCH AND DEVELOPMENT

Bus Scheduling including Dynamic Events

Eleftherios Anagnostopoulos

✉ *eanagnostopoulos@hotmail.com*

GitHub: *https://github.com/pinac0099/dynamic-bus-scheduling*

RESEARCH PROJECT

WITHIN THE FIELD OF

SMART CITY INTERNET OF THINGS APPLICATIONS

Final Report

Abstract

Modern transportation systems should be designed according to the requirements of their passengers, while considering operational costs for the managing organizations, as well as being environmentally friendly. The main objective of this work is to provide a realistic simulation of a transportation system, capable of identifying connections among the road network of operation areas, creating bus lines composed of multiple connected bus stops, simulating travel requests registered by potential passengers, as well as generating routes and timetables for bus vehicles, while taking into consideration factors which could affect the predefined schedule, including unpredictable events (e.g., traffic accidents) or dynamic levels of traffic density.

The implemented bus management system is able to generate timetables dynamically, introducing a reasoning mechanism capable of evaluating travel requests based on dynamic clustering techniques, while offering the opportunity to its administrator to make decisions regarding the number of generated timetables, operating bus vehicles, passengers per timetable, waiting time of passengers, and processing time. In addition, the routes of bus lines are generated or updated dynamically, while taking into consideration real-time traffic data and evaluating parameters, such as covered distance or travelling time, in order to identify the most effective connections between the bus stops of each bus line and make adjustments to the corresponding timetables. Finally, the number of operating bus vehicles that are required in order to transport the passengers of each bus line is estimated, leading to a more efficient distribution of available resources.

Acknowledgements

It is a great opportunity to bestow my heartfelt regards to the people who have been either directly or indirectly involved in the fulfillment of this project.

First and foremost, I would like to express my greatest appreciation to Ms Azadeh Bararsani (Senior Research Engineer in the Data Analytics Field, Ericsson, Stockholm, Sweden) and Ms Aneta Vulgarakis Feljan (Senior Research Engineer in the Data Analytics Field, Ericsson, Stockholm, Sweden) not only for offering me the opportunity to be part of this project, but also for their guidance, patience, and support which allowed me to work efficiently and with high motivation. It is my firm belief that through the engagement with this work, I had the chance to acquire precious experience and enhance my knowledge and skills.

In addition, I would also like to thank all the members of the CityPulse Consortium (University of Surrey (UK), Alexandra Institute (Denmark), Ericsson (Sweden), Siemens (Austria and Romania), National University of Ireland – DERI (Ireland), University of Applied Sciences Osnabrück (Germany), City of Brasov (Romania), City of Aarhus (Denmark), and Kon.e.sis (USA)) for our excellent cooperation.

Last but not least, I am also grateful to Dr Tjark Weber (Senior Lecturer, Uppsala University, Uppsala, Sweden) for accepting the role of reviewer and providing valuable support regarding the evaluation of this research work.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem Definition

There is no doubt that modern transportation systems should be designed according to the requirements of their passengers, while taking into consideration operational costs for the managing companies, as well as being environmentally friendly.

Based on these principles, a competent bus management system should be developed in such a way, ensuring that the following issues are addressed:

- *Dynamic Timetable Generation*: Statically predefined timetables, which might be rarely or never updated, could not be considered as a satisfying option for the passengers. For this reason, a bus management system should comprise reasoning mechanisms capable of identifying and evaluating the travelling preferences of passengers, in order to generate timetables accordingly. Moreover, the generated timetables should also be updated regularly, taking into consideration recently registered travel requests.

- *Dynamic Route Identification*: Daily transportation in urban road networks might include delays due to unpredictable events (e.g., traffic accidents) or high levels of traffic density. As a result, it would be quite useful for a bus transportation system to receive the most recent and accurate information regarding traffic flow, and possess mechanisms capable of processing the input data and identifying the most effective route connections (e.g., the ones with the lowest travelling time) between the bus stops of operation areas.

- *Efficient Use of Vehicle Resources*: Undoubtedly, every administrative company would like to minimize the number of operating bus vehicles which are required in order to transport their passengers, so as to keep operational costs as low as possible. Furthermore, apart from economic costs and despite the fact that modern bus vehicles could be considered as environmentally friendly, keeping their numbers in low levels is of the utmost importance for the environment. In this direction, every bus management system should be in place to ensure that vehicle resources are not wasted unwisely, following the transportation demands of passengers.

1.2 Proposed Solution

Strictly connected with the problem definition, the main objective of this work is to provide a realistic simulation of a transportation system capable of identifying connections among the road network of operation areas, creating bus lines composed of multiple connected bus stops, simulating travel requests registered by potential passengers, as well as generating routes and timetables for bus vehicles, while taking into consideration factors which could affect the predefined schedule, such as unpredictable events (e.g., traffic accidents) or dynamic levels of traffic density.

The implemented bus management system is able to:

- Generate timetables dynamically, introducing a reasoning mechanism capable of evaluating travel requests using dynamic clustering techniques, while offering the opportunity to its administrator to make decisions regarding the number of generated timetables, operating bus vehicles, passengers per timetable, waiting time of passengers, and processing time.
- Generate routes for bus vehicles or make adjustments to the already existing ones dynamically, while processing real-time traffic data and evaluating travelling parameters (e.g., covered distance, travelling time, vehicle speed, or intermediate waypoints), in order to identify the most effective route connections (e.g., the ones with the lowest travelling time) between the bus stops of each bus line and keep the corresponding timetables updated.
- Facilitate the efficient distribution of vehicle resources, estimating the minimum number of operating bus vehicles which are required in order to transport the passengers in each bus line.

1.3 Thesis Outline

Chapter 2 Comprehensive introduction to concepts, algorithms, tools, technologies, and projects relevant to the implemented system.

Chapter 3 Analysis of the implemented system, including information about its architecture and details regarding the implemented components and their interaction.

Chapter 4 Experimental evaluation and comparison with similar projects.

Chapter 5 Conclusions and proposals regarding further enhancements.

Appendix A Description of documents.

Appendix B Pseudocodes.

Appendix C System requirements, deployment guidance, and technical instructions.

Chapter 2

Background and Related Work

This chapter is focused on providing a comprehensive introduction to concepts, algorithms, tools, technologies, and projects relevant to the presented work, in order to facilitate the transition to the following chapters of the report, which contain details regarding the implemented system.

2.1 Concepts and Algorithms

2.1.1 Cluster Analysis

Cluster analysis or *clustering* is the task of separating a set of data objects into groups, known as *"clusters"*, so as the objects which belong to the same cluster are more similar to each other, compared to those in the remaining clusters. It is a widely used technique in many fields of computer science, including machine learning, pattern recognition, image analysis, information retrieval, and data compression.

Despite the existence of a notable number of clustering models (e.g., *connectivity-based* or *hierarchical*, *density-based*, or *distribution-based*), the purpose of this section is to provide some introductory information regarding the *centroid-based* clustering model, so as to be easier for the reader to comprehend the clustering techniques that are applied in the timetable generation algorithm which is introduced in this project.

Centroid-based Clustering

As it is indicated by its name, in *centroid-based* clustering clusters are represented by their centers, which are known as *"centroids"*. The most well-known algorithm was developed by *Stuart P. Lloyd* and is often actually referred to as the ***"k-means algorithm"*** [1]. This algorithm provides an approximate solution to the optimization problem, which includes the identification of k cluster centers and the assignment of each data element to the nearest cluster, so as to minimize the *Euclidean* distance between each element and the corresponding cluster centroid.

More precisely, the *k-means* clustering algorithm includes the following steps of execution:

1. The number k of clusters is provided as input.

2. The starting k "means", which represent the initiatory centroids of the clusters, are initialized. The most commonly used initialization methods are the *Forgy* and *Random Partition* [2]. Using the *Forgy* method, the centroids are randomly selected from the elements of the data set. On the contrary, in the *Random Partition* method each data object is randomly assigned to a cluster before computing the initial centroids.

3. The distance values between the objects of the data set and the initial centroids are calculated, so as to be used while associating each object with the cluster with the nearest centroid.

4. After assigning every object to a cluster, new mean values are calculated corresponding to the new cluster centroids.

5. Steps 3 and 4 are repeated until no alteration is observed, neither on the centroids nor on the assignments (*convergence*).

Regarding computational complexity, even though in the worst case scenario the *k-means* algorithm requires exponential time to converge [3], the smoothed analysis of its complexity is polynomial [4]. As far as its drawbacks are concerned, despite the fact that it is one of the most widely used algorithms for clustering, the definition of the number of clusters in advance is considered as a limitation, since there might exist computational problems demanding dynamic estimation of the number of clusters. For this reason, considerable research efforts have been performed in order to investigate whether the optimal number of clusters could be identified on the run, taking into consideration measurements related to the quality of clustering. For instance, in the *"Dynamic Clustering of Data with Modified K-Means Algorithm"* [5] project the option of selecting a fixed number of clusters or providing a minimum number of clusters as input, which might be increased depending on the quality of clustering, was investigated.

In the current project, a new algorithm for timetable generation is introduced integrating the advantages of the *k-means* clustering algorithm with a non-fixed number of clusters which is not initialized in advance, but is dynamically estimated based on the transportation demands of passengers and a set of parameters (e.g., available vehicle resources or average waiting time of passengers) which are selected by the administrator of the system.

2.1.2 Graph-based Pathfinding

Graph-based pathfinding is the procedure of identifying paths between the nodes of a graph, while taking into consideration their intermediate edges. Undoubtedly, a road network could be represented by a graph, with geographical points as nodes and their corresponding road connections as edges. In this project, pathfinding is applied while retrieving either multiple possible combinations of routes or the most effective ones (e.g., those with the lowest travelling time), connecting the bus stops of selected areas of operation. For this reason, despite the fact that there are various algorithms for pathfinding, this section is focused on presenting the main details of the *Breadth-first*, *Dijkstra's*, and *A-star* search algorithms, since these were used as guidance for the implemented pathfinding algorithms (Subsection 3.4.5).

Breadth-first Search Algorithm

Breadth-first search (BFS) is an algorithm able to identify multiple paths connecting two nodes of a graph, while exploring all the nodes following their intermediate edges. It was invented in 1959 by *Edward F. Moore*, while trying to find the shortest path out of a maze [6], and discovered independently by *C. Y. Lee* in 1961, in the context of routing wires on circuit boards [7]. Apart from identifying multiple paths, using the *breadth-first search* algorithm it is also possible to focus on individual attributes of edges (e.g., distance or travelling time) and select the path which ensures the optimal combination.

The *breadth-first search* algorithm includes the following steps of execution:

1. Starting from the initial node, the neighbor nodes are explored first, before moving to the next level neighbors. For this reason a data storing structure is required, usually a queue or a stack, in order to store the nodes whose neighbors have not yet been explored.
2. For each node of the graph, the cost value is set to infinity and the parent node to null.
3. The initial node of the graph is pushed to the storing structure.
4. A node is retrieved, as long as the number of nodes in the storing structure is above zero. The retrieved node is ignored in case it is marked as "visited". Additionally, the cost value could be used in order to keep priority among the nodes of the structure, in case the algorithm is focused on identifying the optimal path.
5. Following the edges of the retrieved node, the cost and parent values of the its neighbors are updated. Moreover, all the neighbors are pushed to the storing structure and the retrieved node is marked as "visited".
6. If the destination node is retrieved, then the followed path is re-created by checking the parent values of the corresponding nodes.
7. The execution of the algorithm is terminated when there are no more nodes in the storing structure.

As far as the running time complexity of the algorithm is concerned, in the worst-case scenario it could be expressed as $O(n+e)$ [8], where n is the number of nodes and e the number of edges, since every node and edge of the graph should be explored. Additionally, it should be noted that $O(e)$ may vary between $O(1)$ and $O(n^2)$, depending on the density of connections among the nodes of the graph.

Dijkstra's Algorithm

In 1959, *Edsger W. Dijkstra* published an algorithm for connecting two nodes of a graph [9]. Being more specific, the proposed solution is able to find the shortest path between two nodes, by giving priority to the neighbor edges with the lowest distance. Although the original implementation demonstrated $O(n^2)$ as worst-case running time complexity, where n is the number of nodes in the graph, it is possible to improve this measurement at $O(e+nlogn)$, where e is the number of edges [10], taking advantage of data storing structures capable of keeping priority among the nodes according to their distance from the initial node.

The *Dijkstra's* algorithm includes the following steps of execution:

1. Keeping the initial node of the graph as reference point, the distance of the remaining nodes is initially assigned to infinity. Moreover, all the nodes are marked as *"unvisited"* and stored to a set, except for the initial one which is also considered as the *"current"* node.

2. Taking into consideration the edges, which connect the current node with its neighbors, the distance between each one of the unvisited neighbors and the initial node is calculated. If a newly calculated distance value is lower than a previously calculated one, depending on the current followed path, then the previously calculated value is replaced.

3. When all the neighbors of the current node are examined, then the current node is marked as *"visited"* and is removed from the set of unvisited nodes. As a result, it will not be considered again.

4. Selecting from the set of unvisited nodes, the node with the lowest distance value will be marked as current.

5. The execution of the algorithm is terminated when the destination node is marked as visited, indicating that a path between the initial node and the destination is identified, or if there are no nodes left in the unvisited set, indicating that there is no path connecting the initial node with the destination. Additionally, the execution could be terminated in case the number of retrieved routes exceeds a maximum number of routes selected by the administrator.

Taking into consideration the described steps of execution, there is no doubt that the *Dijkstra's* algorithm provides a solution to the problem of identifying the shortest path between a source node and every other node in the graph [11]. On the other hand, the performance of the proposed solution could be improved utilizing *heuristics* to guide the search.

Heuristics

The term *"heuristics"* is used in computer science in order to describe techniques which offer approximate solutions to time-consuming problems, while limiting the execution time in reasonable frames. As a consequence, the selection of a satisfying heuristic method includes the evaluation of the following trade-off criteria:

- *Optimality*: A heuristic function might not be able to identify the optimal, in case several solutions exist for a given problem.

- *Completeness*: In addition, a heuristic might not offer all the possible solutions.

- *Accuracy* and *Precision*: Moreover, the proposed solution might lack in accuracy or precision.

- *Execution Time*: Finally, as it has already been mentioned heuristics are designed for finding quick and approximate solutions, when classic methods are either time-consuming or unable to identify any exact solution. As a result, execution time is a notable factor in the selection of the applicable heuristic function.

A-star Search Algorithm

In 1968, *Peter Hart*, *Nils Nilsson*, and *Bertram Raphael* introduced the *A-star* or *A** search algorithm [12], an extension of the *Dijkstra's* algorithm taking advantage of *heuristics* in order to achieve better performance, while searching for the path with the lowest cost value, connecting two nodes of a graph. The main difference between the two algorithms is observed in the function which is used for calculating the cost value of each followed path. More precisely, considering the last node of the followed path as n, the cost of the followed path is estimated by the following function:

$$f(n) = g(n) + h(n) \tag{2.1}$$

where *g(n)* is the actual cost for travelling from the the starting node to n, and *h(n)* is a heuristic cost estimation for reaching the destination node starting from n.

Apart from the distinctness regarding cost estimation the two algorithms include the same steps of execution, utilizing a data structure for keeping priority among the unvisited nodes according to their cost values, and carrying on execution until either the destination node is marked as visited or there no more unvisited nodes. In fact, the *Dijkstra's* algorithm could be considered as a special case of *A**, where the heuristic function is equal to zero ($h(n) = 0$) for all nodes [13] [14].

Regarding the running time complexity of the *A** search algorithm, it is depended upon the heuristic function. In the worst-case scenario, which includes a search space without limitation, the relation between the number of nodes to be visited and the depth of the selected path is exponential [15]. At this point, it needs to be mentioned that if the search space is unlimited and there is no path connecting the initial node with the destination, then the algorithm will never be terminated. On the contrary, in a more realistic scenario where the search space could be represented by a tree or graph, then the running time complexity of the *A** search algorithm could be polynomial [16].

2.2 Tools and Technologies

2.2.1 CityPulse

An increasing number of cities have started to introduce new *Information and Communication Technology (ICT)* enabled services, with the objective of addressing sustainability as well as improving the operational efficiency of services and infrastructure. In addition, there is increased interest in providing novel or enhanced service offerings and improved experiences to citizens and businesses.

However, a challenge in the smart city approach is integration across different application domains, as well as the engagement of different city departments, city-contracted entrepreneurs and individual enterprises providing services. Today, large amounts of valuable data and sensor information remain unused or are limited to specific application domains due to the large number of specific technologies and formats (traffic information, parking spaces, bus timetables, waiting times at events, event calendars, environment sensors for pollution or weather warnings etc.). Hence, an aggregation of information from various sources is typically done manually and is often out-dated or just static.

CityPulse [17] [18] is a research project with 10 consortium members focused on developing, building, and testing a distributed framework for the semantic discovery and processing of large-scale real-time *Internet of Things (IoT)* and relevant social data streams for knowledge extraction in urban environments.

To achieve this objective, the project has developed a large set of software components, architecture and best practices, use-case scenarios and demonstrators, integrating dynamic data sources and context-dependent on-demand adaptations of processing chains during run-time. The developed tools and components aim to bridge the gap between the application technologies on the IoT and real world data streams. Finally, the provided framework includes middleware, common interfaces and semantic models, as well as different components and processing methods that enable smart city applications using human and machine sensory data.

Integration with the CityPulse Data Bus Component

The implemented system is integrated with the *CityPulse Data Bus* component, which is used in the *CityPulse* framework in order to share among components semantically annotated datasets of real traffic events [19] [20], collected by fixed sensors in the city infrastructure (e.g., streets, public buildings, utility systems) or in personal and business properties (e.g., vehicles, homes, buildings).

2.2.2 OpenStreetMap

Introduced by *Steve Coast* in 2004 and motivated by restrictions regarding the availability of geographic information across the world, *OpenStreetMap (OSM)* [21] is a collaborative project towards the creation of a free and editable map of the world.

Data Format

In *OpenStreetMap*, a topological data structure is utilized in order to represent the provided map, consisting of the following core elements:

- Node: A single point in space defined by its geographic coordinates (latitude and longitude values) and a unique id (*osm_id*). Nodes can be used on their own to define point features. When used in this way, a node will normally have at least one tag to define its purpose. Nodes may have multiple tags and/or be part of a relation. For example, the following tag ("amenity=telephone") could be included in a node, representing a telephone box.

- Way: Ordered list of nodes representing linear features (e.g., pedestrian or cycling paths, streets, or rivers) or enclosed filled areas of territory (e.g., parking areas, parks, forests, or lakes). A way normally contains at least one tag and could also be included within a relation. Finally, ways are divided into *open* or *closed*, depending on the connection between the last and first node of the way.

- Tag: Key-value pairs used in order to describe specific map elements (i.e., nodes, ways, or relations), providing details such as type, name, or physical properties. Some examples of tags could be: "highway=residential", "name=Park Avenue", or "maxspeed=50".

- Relation: One of the core data elements that consists of one or more tags and also an ordered list of one or more nodes, ways and/or relations as "members" which is used to define logical or geographic relationships between other elements. A member of a relation can optionally have a "role" which describes the part that a particular feature plays within a relation. Some examples of relations could include turn restrictions or routes consisting of multiple ways.

In this work, *OpenStreetMap* is used in order to provide information regarding the road network of operation areas, including bus stops, road connections, type of roads, and speed limits. The main reason for the selection of *OpenStreetMap* is the lack of restrictions concerning usage rights, which offers the opportunity of testing the implemented system without any kind of limitation.

2.2.3 MongoDB

MongoDB [22] is a free and open-source cross-platform document-oriented *NoSQL* database, designed for ease of development and scaling. In MongoDB, data is stored in *BSON* [23] *documents* which are *JSON*-style [24] data structures. Documents contain one or more fields, and each field contains a value of specific data type, including arrays, binary data, and sub-documents. Documents that tend to share a similar structure are organized as *collections*.

In this work, *MongoDB* was selected as development tool for the database of the implemented system, based on its ability to excel in use cases where relational databases are not a good fit, like applications with large volumes of rapidly changing structured, semi-structured, unstructured, and polymorphic data, as well as applications with large scalability requirements or multi-data center environments. These properties could be quite useful while developing the database of a transportation system containing multiple and rapidly changing entries of routes and timetables, as well as travel requests from thousands of users.

2.2.4 Gunicorn

Gunicorn or *"Green Unicorn"* [25] is a Python [26] WSGI HTTP [27] server, based on the pre-fork worker model, broadly compatible with various Web frameworks, simply implemented, light on server resources, and fairly speedy. In this project, *Gunicorn* is used in order to provide the Web server functionality of the Route Generator, a stand-alone Web server implementation responsible for identifying routes, for bus vehicles, between the bus stops of operation areas.

Web Server Gateway Interface (WSGI)

Gunicorn is implemented according to the principles of the *Web Server Gateway Interface (WSGI)* [28], a specification for a standard and universal interface between Web servers and Python Web applications or frameworks, focused on promoting Web application portability across a variety of Web servers. It was originally specified in the *Python Enhancement Proposal (PEP)* 333, authored by *Phillip J. Eby*, and published on 7 December 2003.

According to its standards, the *WSGI* includes the following interacting sides:

1. The “server” or “gateway” which is responsible for establishing communication with the application and providing environment information as well as a callback function to the the application side.

2. The “application” or “framework” which is focused on processing requests and returning responses to the server side, utilizing the provided callback function.

3. In addition, there might also exist a middleware facilitating communication between the two aforementioned sides.

Pre-fork Worker Model

Gunicorn is based on the pre-fork worker model, including a central master process in charge of initiating and managing a set of synchronous or asynchronous worker processes, which are responsible for handling requests registered by individual clients.

2.3 Related Work

2.3.1 Mobile Network Assisted Driving

Mobile Network Assisted Driving (MoNAD) [29] is a project of *Ericsson Research* implemented in collaboration with *Uppsala University*, focused on generating transportation schedules for a bus management system. Being more specific, the implemented system includes two *Android* [30] applications, used by users and drivers respectively, as well as a number of back-end components used in order to receive travel requests, identify bus routes, and generate timetables, using a genetic algorithm. Taking advantage of the provided *Android* applications, users have the opportunity to make searches, register travel requests, and receive travel recommendations for trips that they might be interested in. On the other hand, bus drivers are able to receive details about the route of each bus vehicle and the number of passengers at each bus stop.

The current project could be considered as an extension or alternative of *MoNAD*, introducing a new algorithm for evaluating travel requests and generating timetables, as well as providing features such as real-time traffic flow detection and dynamic route identification, based on input parameters (e.g., levels of traffic density).

2.3.2 Mobile Phone Based Participatory Sensing

In 2013, a scientific group from the *Nanyang Technological University* of Singapore published an article with title *“How Long to Wait?: Predicting Bus Arrival Time with Mobile Phone based Participatory Sensing”* [31] regarding the development of a prototype application, able to predict the arrival time of a bus vehicle based on participatory sensing of passengers. Taking advantage of commodity mobile phones, the bus passengers’ surrounding environmental context is effectively collected and utilized to estimate the bus travelling routes and predict bus arrival time at various bus stops. The implemented system solely relies on the collaborative effort of the participating users and is independent from the bus operating companies, so it can be easily adopted to support universal bus service systems without requesting support from particular bus operating companies.

Chapter 3

Implementation Analysis

3.1 System Description

The main objective of this work is to provide a realistic simulation of a bus transportation system and introduce a reasoning mechanism capable of evaluating travel requests and generating timetables for bus vehicles, while offering the option to its administrator to make decisions regarding the number of generated timetables, operating bus vehicles, number of passengers per vehicle, average waiting time of passengers, and processing time. In addition, traffic flow detection is utilized in order to make adjustments to the regular path of each bus vehicle and limit the waiting time of passengers, which could be increased due to traffic congestion.

Being more specific, as illustrated in the architecture diagram of Figure 3.1, the implemented system is able to:

- (OpenStreetMap Parser): Process OpenStreetMap files and extract geospatial data related to the road network of operation areas (e.g., bus stops and parameters of intermediate road connections).
- (Route Generator): Identify all the possible route connections between the bus stops of operation areas, implementing a variation of the *Breadth-first* search algorithm.
- (Look Ahead): Generate bus lines connecting the bus stops of operation areas.
- (Traffic Data Parser): Receive real-time traffic data from the CityPulse Data Bus.
- (Traffic Data Simulator): Simulate traffic events capable of affecting the normal schedule.
- (Travel Requests Simulator): Simulate travel requests registered by potential passengers.
- (Route Generator): Identify the less time-consuming routes connecting the bus stops of operation areas, implementing a variation of the $A*$ search algorithm, while taking into consideration current levels of traffic density.
- (Look Ahead): Apply a timetable generation algorithm capable of evaluating travel requests, utilizing dynamic clustering procedures, and generating timetables for bus vehicles, while offering the option to its administrator to make decisions regarding the number of generated timetables, operating bus vehicles, number of passengers per vehicle, average waiting time of passengers, and processing time.

- (Look Ahead): Monitor the levels of traffic density and make adjustments to the regular path of each bus, in order to limit the level of affection on the average waiting time of passengers due to traffic incidents.

3.2 System Architecture

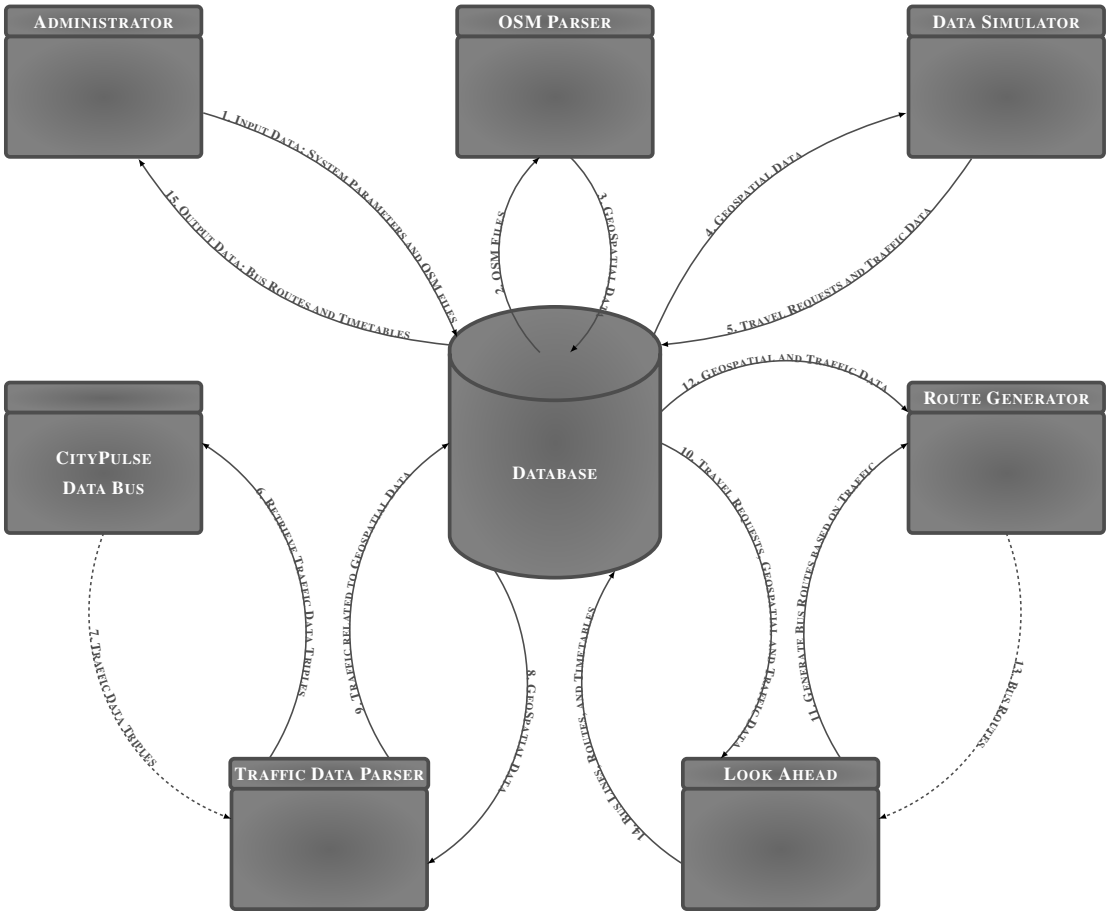

Figure 3.1: System Architecture Diagram.

3.3 Interaction of Components

3.3.1 Parsing of Geospatial Data

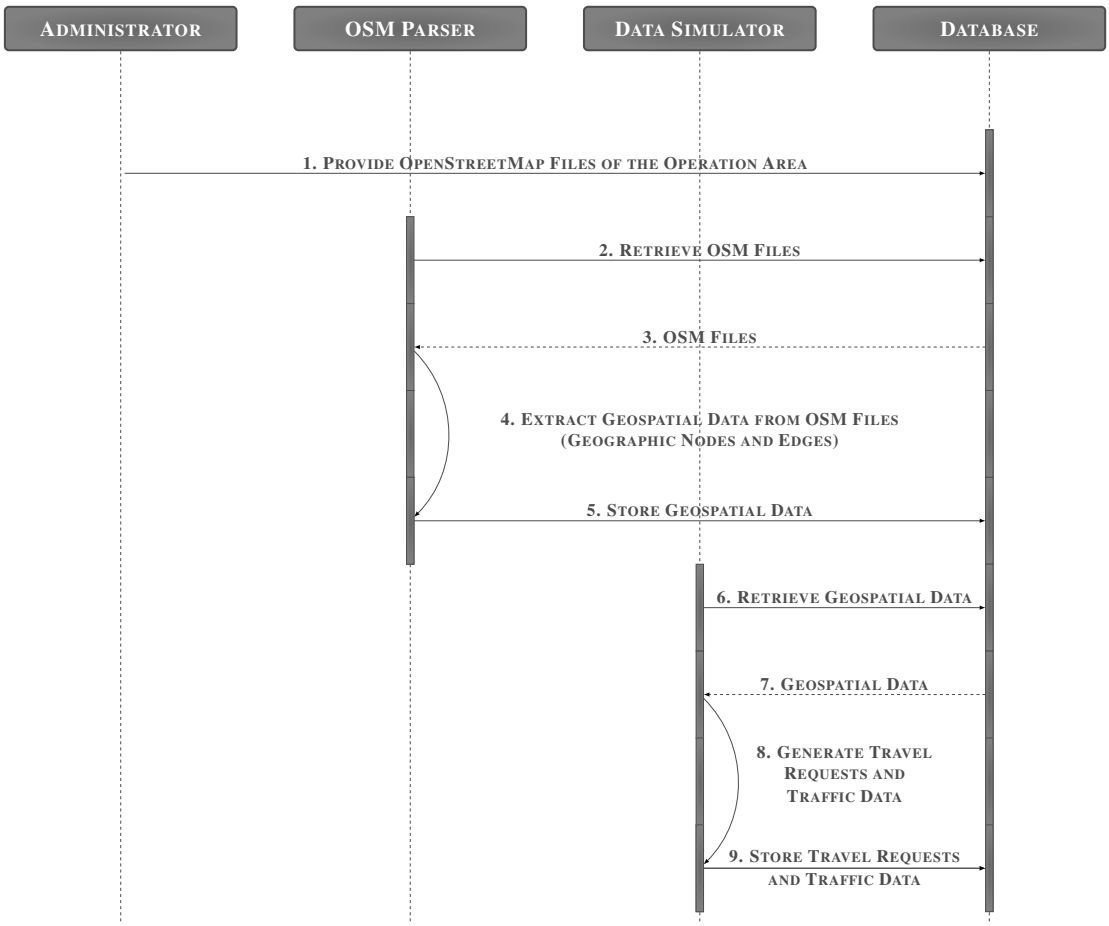

Figure 3.2: Parsing of Geospatial Data Sequence Diagram.

As illustrated in the sequence diagram of Figure 3.2, the following interaction steps are performed between the System Administrator, OpenStreetMap Parser, Data Simulator, and System Database:

1. The System Administrator is responsible for providing the OpenStreetMap files, which include information about the road network and infrastructure of the operation area.

2-5. The provided OpenStreetMap files are retrieved and processed by the OpenStreetMap Parser, which extracts data regarding the geographical nodes of the selected area (e.g., buildings, bus stops, or geographical points), as well as the road connections (edges) which connect them. The extracted geospatial data is stored at the corresponding collections of the System Database (Address, BusStop, Edge, Node, Point, and Way).

6-9. A connection is established between the Data Simulator and the System Database so as the stored bus stop and edge documents to be retrieved. The bus stop documents are used by the Travel Requests Simulator while generating new travel requests, while the Traffic Data Simulator updates the traffic density values of the edge documents.

3.3.2 Parsing of Traffic Data

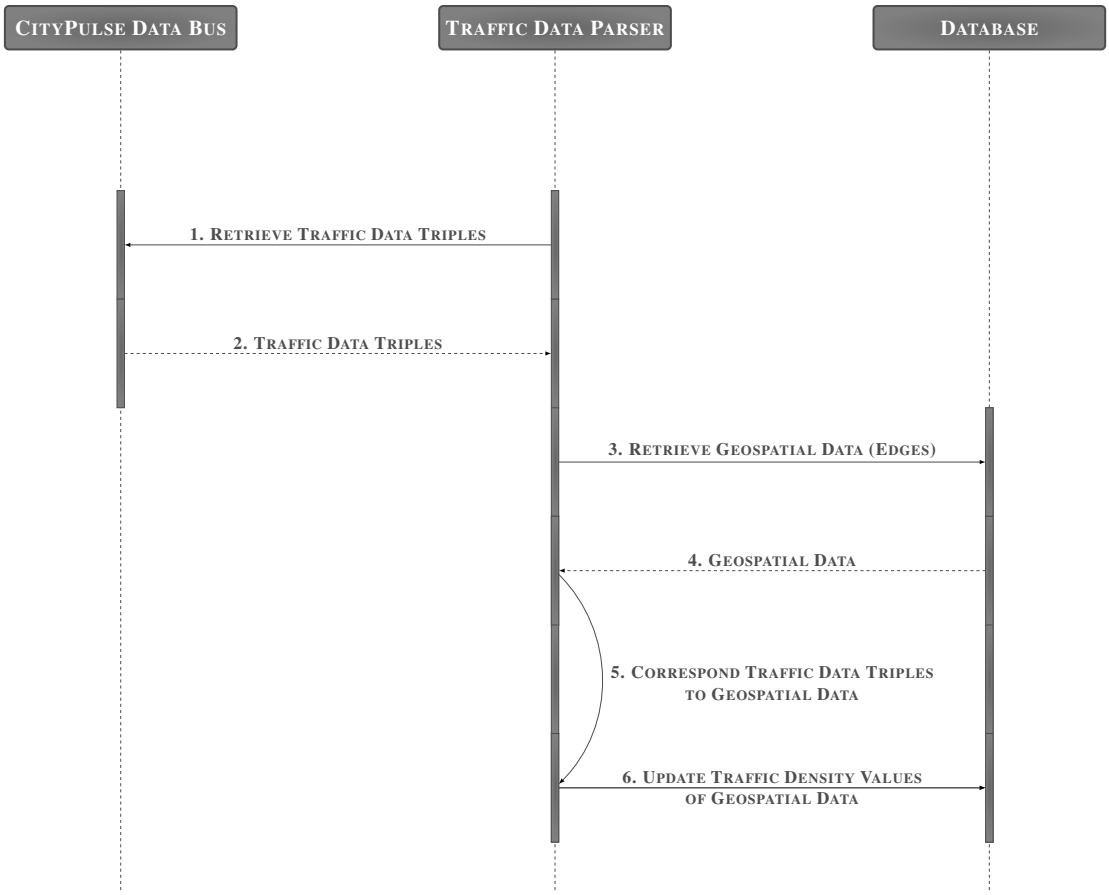

Figure 3.3: Parsing of Traffic Data Sequence Diagram.

As illustrated in the sequence diagram of Figure 3.3, the following interaction steps are performed between the CityPulse Data Bus, Traffic Data Parser, and System Database:

1-2. The Traffic Data Parser connects to the CityPulse Data Bus in order to retrieve the triples, which contain information about the traffic events of the operation area.

3-4. A connection is established between the Traffic Data Parser and the System Database so as the stored edge documents to be retrieved. The documents were stored by the OpenStreetMap Parser while extracting geospatial data from the provided OpenStreetMap files.

5-6. Taking into consideration the geographical coordinates of the retrieved traffic events, the Traffic Data Parser identifies the edge documents which are affected and updates their corresponding traffic density values.

3.3.3 Timetable Generation

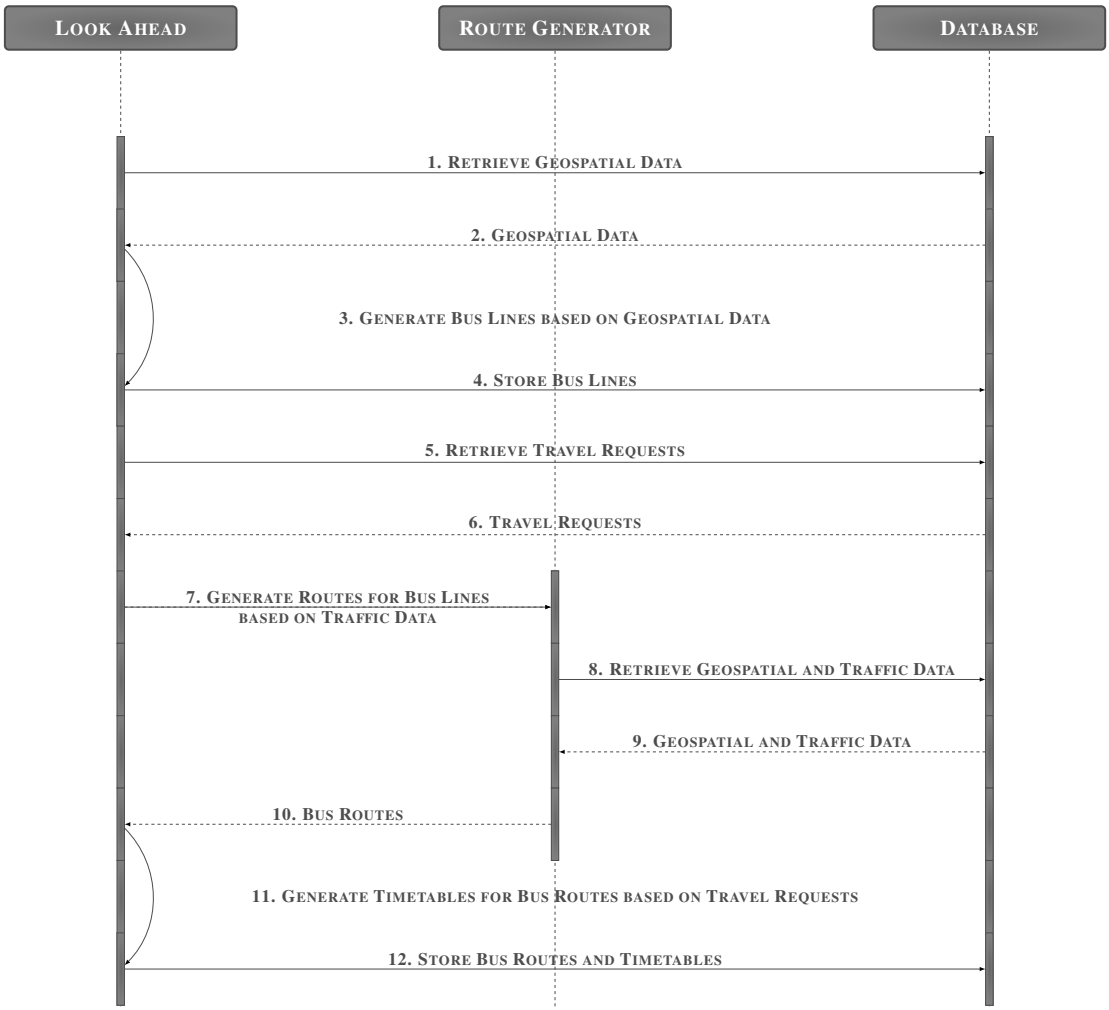

Figure 3.4: Timetable Generation Sequence Diagram.

As illustrated in the sequence diagram of Figure 3.4, the following interaction steps are performed between the Look Ahead, Route Generator, and System Database:

1-2. A connection is established between the Look Ahead and the System Database. The Look Ahead retrieves the bus stop documents which were stored by the OpenStreetMap Parser.

3-4. The Look Ahead generates the bus line documents of the system, based on the retrieved bus stop documents, and stores them to the corresponding collection of the System Database.

5-6. The Look Ahead retrieves the travel request documents with departure datetimes between specified datetime periods, which are selected by administrator of the system. The main objective of the timetable generation algorithm is to generate timetables leading to the minimum possible average waiting time for these travel requests, while considering limitations such as the number of available bus vehicles or delays from traffic density.

7-10. A request is sent from the Look Ahead to the Route Generator in order to generate the less time-consuming routes for the bus lines of the system, while taking into consideration the

current levels of traffic density. The Route Generator connects to the System Database in order to retrieve the stored edge documents, which include data about route connections and traffic density. Based on the retrieved edge documents, the less time-consuming routes which connect the bus lines of the system are identified.

11-12. The Look Ahead evaluates the retrieved travel request documents, while taking into consideration the parameters of generated bus routes (e.g., travelling time), and generates the timetables the system. Finally, the generated bus routes and timetables are stored to the corresponding collections of the System Database.

3.3.4 Timetable Update

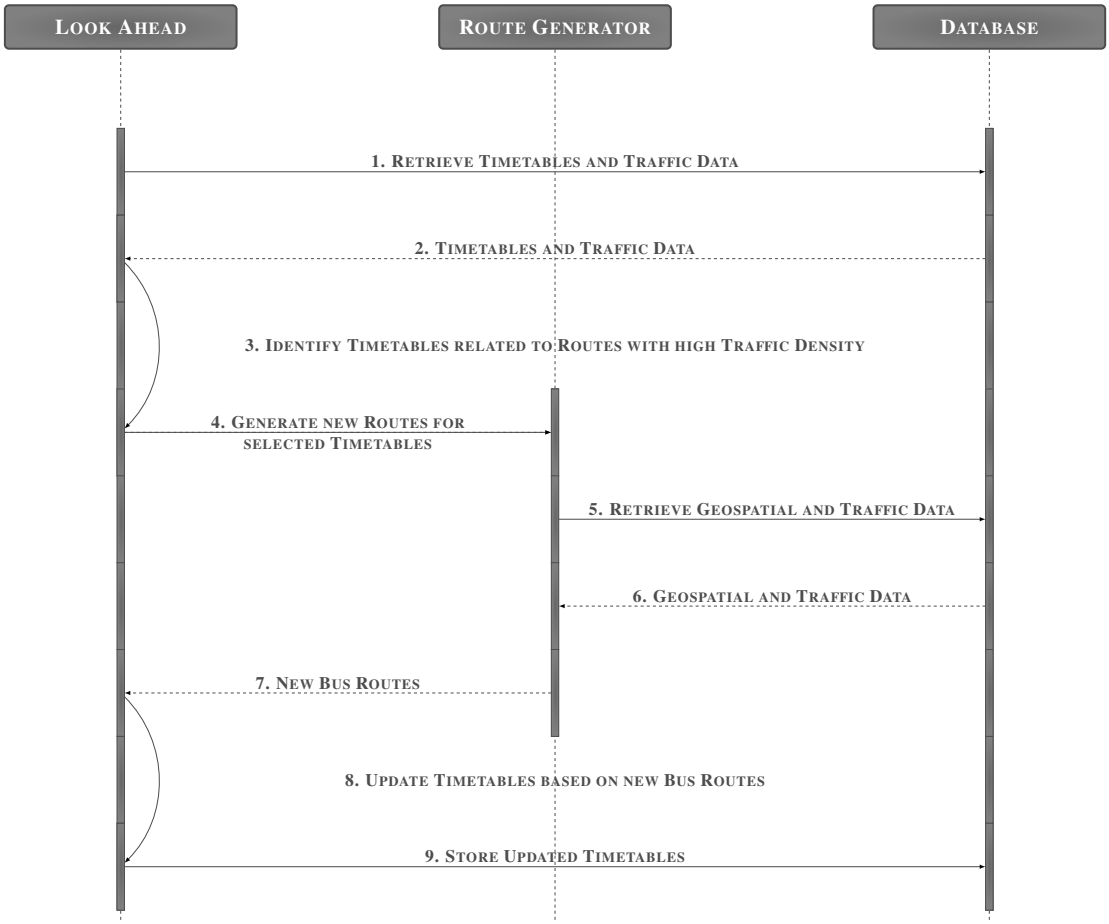

Figure 3.5: Timetable Update Sequence Diagram.

As illustrated in the sequence diagram of Figure 3.5, the following interaction steps are performed between the Look Ahead, Route Generator, and System Database:

1-2. The Look Ahead connects to the System Database in order to retrieve the stored timetable and edge documents. As it has already been mentioned, the edge documents include data about the current levels of traffic density.

3. While processing the retrieved documents, the Look Ahead identifies the bus routes with increased traffic density and the timetables which are related to them.

4. A request is sent from the Look Ahead to the Route Generator is order to generate new bus routes, connecting the same bus stops, for the bus routes with high levels of traffic density.

5-6. The Route Generator connects to the System Database in order to retrieve the stored edge documents.

7. The Route Generator generates new bus routes, taking into consideration the current levels of traffic density, and responds to the request of the Look Ahead.

8-9. The response of the Route Generator is processed by the Look Ahead and the corresponding timetable documents are updated.

3.4 Component Description

3.4.1 System Database

The System Database is a *MongoDB* implementation consisting of the following collections:

- AddressDocuments
- BusLineDocuments
- BusStopDocuments
- BusStopWaypointsDocuments
- BusVehicleDocuments
- EdgeDocuments
- NodeDocuments
- PointDocuments
- TimetableDocuments
- TrafficEventDocuments
- TravelRequestDocuments
- WayDocuments

AddressDocuments Collection

An address document is used in order to store data about an address and contains the following fields:

- *_id*: The MongoDB *id* of the document.
- *name*: The name of address.
- *node_id*: The OpenStreetMap *id* of the node to which the address corresponds.
- *point*: The geographical point of the address, consisting of a pair of coordinates (latitude, longitude).

BusLineDocuments Collection

A bus line document is used in order to store data about a bus line and contains the following fields:

- *_id*: The MongoDB *id* of the document.
- *bus_line_id*: The *id* of the bus line.
- *bus_stops*: A list containing the bus stops of the bus line.

BusStopDocuments Collection

A bus stop document is used in order to store data about a bus stop and contains the following fields:

- *_id*: The MongoDB *id* of the document.
- *osm_id*: The OpenStreetMap *id* of the bus stop.
- *name*: The name of the bus stop.
- *point*: The geographical point of the bus stop, consisting of a pair of coordinates (latitude, longitude).

BusStopWaypointsDocuments Collection

A bus stop waypoints document is used in order to store data about all the possible combinations of edge documents, connecting a pair of bus stops, and contains the following fields:

- *_id*: The MongoDB *id* of the document.
- *starting_bus_stop*: The document of the starting bus stop.
- *ending_bus_stop*: The document of the ending bus stop.
- *waypoints*: A list including lists of *ids* of the connected edge documents.

BusVehicleDocuments Collection

A bus vehicle document is used in order to store data about a bus vehicle and contains the following fields:

- *_id*: The MongoDB *id* of the document.
- *bus_vehicle_id*: The *id* of the bus vehicle.
- *maximum_capacity*: The maximum number of passengers that could be transported by the bus vehicle.
- *routes*: A list containing information about the routes – timetables of the bus vehicle.

EdgeDocuments Collection

An edge document is used in order to store data about a road connection between two nodes or geographical points and contains the following fields:

- *_id*: The MongoDB *id* of the document.
- *starting_node*: The starting node or point of the edge.
- *ending_node*: The ending node or point of the edge.
- *max_speed*: The maximum allowed speed for this road.
- *road_type*: The type of road.
- *way_id*: The *id* of the way to which the edge belongs.
- *traffic_density*: A value between 0 and 1, used as traffic density measurement.

NodeDocuments Collection

A node dcument is used in order to store data about a node and contains the following fields:

- *_id*: The MongoDB *id* of the document.
- *osm_id*: The OpenStreetMap *id* of the node.
- *tags*: A dictionary containing the tags of the node.
- *point*: The geographical point of the node, consisting of a pair of coordinates (latitude, longitude).

PointDocuments Collection

A point document is used in order to store data about a point and contains the following fields:

- *_id*: The MongoDB *id* of the document.
- *osm_id*: The OpenStreetMap *id* of the point.
- *point*: The pair of coordinates (latitude, longitude) of the point.

TimetableDocuments Collection

A timetable document is used in order to store data about the route of a bus vehicle and contains the following fields:

- *_id*: The MongoDB *id* of the document.
- *timetable_id*: The *id* of the timetable.
- *bus_line_id*: The *id* of the bus line to which the timetable belongs.
- *bus_vehicle_id*: The *id* of the bus vehicle that the timetable corresponds to.
- *timetable_entries*: A list containing details about the followed route, such as starting and ending bus stops, departure and arrival datetimes, and number of passengers.
- *travel_requests*: The travel requests which are served by the timetable.

TrafficEventDocuments Collection

A traffic event document is used in order to store data about a traffic event and contains the following fields:

- *_id*: The MongoDB *id* of the document.
- *event_id*: The *id* of the traffic event.
- *event_type*: The type of the traffic event.
- *event_level*: An integer value between 1 and 5, used in order to describe the level of the traffic event.
- *point*: The geographical point of the traffic event, consisting of a pair of coordinates (latitude, longitude).

TravelRequestDocuments Collection

A travel request document is used in order to store data about a travel request and contains the following fields:

- *_id*: The MongoDB *id* of the document.
- *client_id*: The *id* of the client/passenger.
- *bus_line_id*: The *id* of the bus line to which the travel request corresponds.
- *starting_bus_stop*: The starting bus stop of the travel request.
- *ending_bus_stop*: The ending bus stop of the travel request.
- *departure_datetime*: The departure datetime preference of the client.
- *arrival_datetime*: The estimated arrival datetime of the passenger.
- *starting_timetable_entry_index*: A parameter used in order to identify the index of the starting timetable entry of the travel request.
- *ending_timetable_entry_index*: A parameter used in order to identify the index of the ending timetable entry of the travel request.

WayDocuments Collection

A way document is used in oder to store data about a way and contains the following fields:

- *_id*: The MongoDB *id* of the document.
- *osm_id*: The OpenStreetMap *id* of the way.
- *tags*: A dictionary containing the tags of the way.
- *references*: A list including the OpenStreetMap *ids* of the nodes and points that comprise the way.

3.4.2 OpenStreetMap Parser

Based on *import.parser* [32] which is a Python library for parsing of OpenStreetMap data, the OpenStreetMap Parser is a component responsible for processing the provided OpenStreetMap files and extracting geospatial data related to the road network and infrastructure of operation areas.

More precisely the developed component is able to parse the following types of data:

- **Node**: A dictionary containing an *osm_id*, a dictionary of tags, and a pair of coordinates (latitude, longitude). Nodes are used in OpenStreetMap files in order to represent objects (e.g., geographical points, buildings, bus stops, traffic lights, telephone boxes etc.).
- **Point**: A dictionary containing an *osm_id* and a pair of coordinates (latitude, longitude), used in order to represent a geographical point.
- **Bus Stop**: A dictionary containing an *osm_id*, a name, and a pair of coordinates (latitude, longitude), used in order to represent a bus stop. A bus stop is also a node containing one of the following nodes: ("bus"='yes", "highway"="bus_stop", "public_transport"= "platform", "public_transport"="stop_area").
- **Address**: A dictionary containing a name, the *osm_id* of the corresponding node (*node_id*), and a pair of coordinates (latitude, longitude), used in order to represent an address.
- **Way**: A dictionary containing an *osm_id*, a dictionary of tags, and a list of references (*osm_id* values) corresponding to the connected points and nodes that comprise the way.
- **Edge**: A dictionary containing a starting and ending node or point, the maximum allowed speed, type of road, *osm_id* of the way to which the edge belongs, and a value between 0 and 1 used as traffic density measurement. Edges are parsed from ways and represent road connections between nodes and points.

Finally, as illustrated in the sequence diagram of Figure 3.2, the OpenStreetMap Parser is also capable of connecting to the System Database in order to store the extracted geospatial data in the corresponding collections (Address, BusStop, Edge, Node, Point, and Way).

3.4.3 Data Simulator

The Data Simulator is a component responsible for providing data, which is used for testing purposes, and comprises the Travel Requests Simulator and the Traffic Data Simulator.

Travel Requests Simulator

The Travel Requests Simulator component is used for generating travel request documents, which would be registered by potential passengers, and populating the corresponding collection of the System Database. Utilizing the Travel Requests Simulator, the administrator of the system could make decisions regarding the distribution of travel requests, as well as affecting the number of generated documents according to the datetime of registration. In this way, it would be possible to provide a realistic simulation of transportation demand, since more travel requests would be generated for periods with high demand (e.g., 7am - 9am or 4pm - 6pm) and less documents in periods when demand is not so high (e.g., 1am - 5am).

Traffic Data Simulator

The Traffic Data Simulator component is used in order to simulate traffic events, which could possibly affect the normal schedule of the system and increase the average waiting time of passengers. Being more specific, the Traffic Data Simulator is able to connect to the System Database, retrieve documents for the EdgeDocuments Collection which correspond to selected bus lines or connect specific bus stops, and modify their traffic density values. Following a similar approach with the Travel Requests Simulator, the Traffic Data Simulator is able to generate higher or lower traffic density values, depending on the datetime of the registered traffic event.

3.4.4 Traffic Data Parser

As illustrated in the sequence diagram of Figure 3.3, the main task of the Traffic Data Parser is to establish a connection with the CityPulse Data Bus, so as to retrieve traffic events and populate the TrafficEventDocuments Collection of the System Database. An additional assignment of the Traffic Data Parser is to determine the relation between the retrieved traffic events and the documents which are already stored at the EdgeDocuments Collection, by comparing their corresponding longitude and latitude values. In this way, traffic events become related to the edges which connect the nodes of the operation area, and traffic density values are updated depending on the output of the CityPulse Data Bus.

3.4.5 Route Generator

In this project, each *"route"* is represented as a list of connected edge documents, leading from one geographical point to another. The Route Generator is a component responsible for identifying and evaluating routes, for bus vehicles, connecting the bus stops of operation areas. In this way, it is possible to identify the route with the lowest cost value (e.g., travelling time, covered distance, or traffic density) or retrieve multiple possible routes, which are called as *"waypoints"* and are represented by a list of alternative routes, connecting two or more bus stops.

Web Server Functionality

Based on *Gunicorn* and designed according to the principles of the *Web Server Gateway Interface (WSGI)*, the Route Generator is a stand-alone HTTP server implementation which can serve multiple clients concurrently, offering support for the following requests:

get_route_between_two_bus_stops

Description: Identification of the route with the lowest cost value, connecting two provided bus stops.

Parameters: A starting and an ending bus stop (or the corresponding bus stop names).

Response (Subsection A.2.1): Contains information about the starting and ending bus stop, covered distance, required travelling time, intermediate nodes, and the edges which connect them.

get_route_between_multiple_bus_stops

Description: Identification of the route with the lowest cost value, connecting a list of provided bus stops.

Parameters: A list of bus stops (or the corresponding bus stop names).

Response (Subsection A.2.2): Consisting of a list of routes among the intermediate bus stops, including information about starting and ending bus stops, covered distance, required travelling time, intermediate nodes, and the edges which connect them.

get_waypoints_between_two_bus_stops

Description: Identification of multiple possible routes connecting two provided bus stops. The number of retrieved routes could be limited by the administrator of the system.
Parameters: A starting and an ending bus stop (or the corresponding bus stop names).
Response (Subsection A.2.3): Consisting of a double list containing multiple possible routes connecting the two provided bus stops, enclosing information about intermediate nodes, maximum allowed speed for bus vehicles, type of crossed roads, and current levels of traffic density.

get_waypoints_between_multiple_bus_stops

Description: Identification of multiple possible routes, connecting a list of provided bus stops. The number of retrieved routes could be limited by the administrator of the system.
Parameters: A list of bus stops (or the corresponding bus stop names).
Response (Subsection A.2.4): Following a similar format with the response of the previous request, it includes a double list containing multiple possible routes connecting the provided bus stops, enclosing information about intermediate nodes, maximum allowed speed for bus vehicles, type of crossed roads, and current levels of traffic density.

Route Identification and Evaluation

The Route Generator is capable of identifying and evaluating route connections between two or more selected bus stops, applying a variation of the $A*$ Search Algorithm. More precisely, it is possible to retrieve the route with the lowest cost value, where the cost of each followed path is calculated while taking into consideration parameters which are selected by the administrator of the system, such as the required travelling time, covered distance, or traffic density.

Regarding its **inputs**, the implemented algorithm receives a list of bus stops or their corresponding names. If the provided list contains more than two bus stops or names, then the routes with the lowest cost connecting the intermediate starting and ending bus stop combinations are identified, so as to create the route which connects the first with the last bus stop. Moreover, the algorithm comprises the following **steps of execution**:

1. A connection is established between the Route Generator and the System Database, in order to retrieve all the documents of the EdgeDocuments Collection, containing detailed information about road connections among the nodes of the provided areas of operation. In addition, the corresponding documents of the BusStopDocuments Collection are retrieved, in case only the bus stop names were provided as inputs.

2. According to the standards of $A*$ Search Algorithm, priority is given to the neighbor nodes which ensure the lowest possible cost for the followed path. For this reason, a data storing structure is used in order to store the nodes whose neighbors have not yet been explored, while keeping priority depending on the corresponding cost values.

3. The cost value for each node is calculated by adding two values. The first one represents the real cost for travelling from the initial node to the current one, taking into consideration the road details of intermediate edges, such as distance, speed limits, type of roads, and delays due to the current levels of traffic density. On the other hand, the second one is a heuristic estimation, focusing on distance, which offers a prediction about the cost of travelling form the current node to the last one.

4. The cost value of the initial node is calculated and the node is pushed into the data storing structure.

5. The node with the lowest cost value is retrieved from the data storing structure, as long as the number of stored elements is greater than zero. The cost values of its neighbors are calculated, taking into consideration the cost value of the current node and the road parameters of the edges which connect the current node with its neighbors. For each neighbor, the lowest cost value and the corresponding followed path are stored, replacing greater cost values in case the node was considered earlier. In addition, all the neighbors are pushed into the data storing structure, in case they have not been already pushed, in order to allow their corresponding neighbors to be considered.

6. The execution of the algorithm is terminated when the destination node is retrieved from the data storing structure, indicating that the less costly path between the desired nodes is identified, or if there are no more nodes to be explored, meaning that there is no path connecting the initial node with the destination.

The **output** of the algorithm provides information about covered distance and required travelling time among the nodes of the identified path, as well as details for the intermediate point and edge documents.

As far as **running time complexity** is concerned, keeping in mind that the heuristic function has no major effect on the complexity of the algorithm, since it includes a simple calculation based on the geographical distance of nodes, it is similar with the running time complexity of the *Dijkstra's* algorithm. More precisely, in the worst-case scenario the algorithm demonstrates $O(e+nlogn)$ running time complexity, where n is the number of nodes in the graph and e the number of edges, taking advantage of the data storing structure which is capable of keeping priority among the nodes according to their cost values.

Finally, the route identification and evaluation algorithm could be described by the **pseudocode** of Section B.1.

Waypoints Identification

The Route Generator is also able to identify all the possible routes connecting a list of providing bus stops, implementing an alteration of the *Breadth-first* Search Algorithm, while offering the opportunity to the administrator of the system to limit the number of retrieved routes, since there might be infinitely many possible results.

More precisely, the implemented algorithm requires a list of bus stops, or their corresponding names, as **inputs**. In case the provided list contains more than two bus stops or names, then

all the possible routes connecting the intermediate starting and ending bus stop combinations are identified, in oder to form the routes which connect the first bus stop with the last one. In addition, the algorithm includes the following **steps of execution**:

1. A connection is established between the Route Generator and the System Database, in order to retrieve all the documents of the EdgeDocuments Collection, containing detailed information about road connections among the nodes of the provided areas of operation. Moreover, the corresponding documents of the BusStopDocuments Collection are retrieved, in case only the bus stop names were provided as inputs.

2. Following the principles of *breadth-first* search, the neighbors of each node are explored first, before moving to the next level neighbors. For this reason, a data storing structure is utilized in order to store the nodes whose neighbors have not yet been explored. Furthermore, each node contains a double list in order to keep track of all the followed paths leading to it.

3. The initial node is pushed into the data storing structure.

4. A node is retrieved from the data storing structure, as long as the number of stored nodes is above zero. The retrieved node is ignored in case it is marked as "visited". Otherwise, following its edges, the corresponding neighbors are identified and pushed into the structure, while adding the current node to the corresponding lists representing their followed paths. Moreover, the current node is marked as "visited".

5. If the destination node is retrieved, corresponding to the last bus stop, then all the possible path connecting the first and last node are re-created, checking the followed paths of the intermediate nodes.

6. The execution of the algorithm is terminated in case there are no more nodes in the storing structure or the number of retrieved routes has reached the input of the administrator.

The **output** of the algorithm includes a list containing the alternative routes which connect the first with the last node. As it has already been mentioned, each route is represented by a list of edge documents, connecting the intermediate nodes of the route and including details about the its type, maximum allowed speed for bus vehicles, as well as information about the current levels of traffic density.

Regarding the **running time complexity** of the algorithm, similarly with the *Breadth-first* Search Algorithm, in the worst-case scenario it could be expressed as $O(n+e)$, where n is the number of nodes and $|e|$ the number of edges, since every node and edge of the graph should be explored. Additionally, $O(e)$ may vary between $O(1)$ and $O(n^2)$, depending on the density of edges connecting the nodes of operation areas.

Finally, the waypoints identification algorithm could be described by the **pseudocode** of Section B.2.

3.4.6 Look Ahead

The Look Ahead is a component responsible for generating the bus lines of the system, connecting the bus lines to routes provided by the Route Generator, producing timetables for bus

lines while evaluating travel requests and distributing the available bus vehicle resources, and updating the existing timetables while taking into consideration modifications in traffic density.

Bus Line Generation

A detailed description of a bus line document is provided in Subsection A.1.2.

The bus line generation algorithm requires the following **inputs**:

- *bus_line_id*: The *id* of the bus line which is going to be generated.
- *bus_stop_names*: A list containing the names of bus stops of the bus line.

The bus line generation algorithm includes the following **steps of execution**:

1. A connection is established between the Look Ahead component and the System Database, so as the list of bus stops which correspond to the provided names to be retrieved. The execution of the algorithm is interrupted and no bus line is generated, in case there is a name which does not correspond to a stored bus stop.
2. While generating a bus line, the intermediate waypoints of the bus routes which are generated while combining starting and ending bus stops of the bus line, should be stored as bus stop waypoint documents in the System Database. The Look Ahead component evaluates the existing documents and communicates with the Route Generator in order to identify the waypoints of the bus routes which are not already stored. The newly generated documents are getting stored to the corresponding collection of the System Database. The execution of the algorithm is interrupted and no bus line is generated, in case the Route Generator is not able to identify a possible route between the provided bus stops.
3. The newly generated bus line document is getting stored to the corresponding collection of the System Database. In case there is an already existing document, with the same *bus_line_id*, then the list of bus stops of the existing document is updated.

The bus line generation algorithm produces the following **outputs**:

- The generated bus line document.
- The generated bus stop waypoint documents, which include include information about all the possible route connections among the bus stops of the bus line.

Timetable Generation

A detailed description of a timetable document is provided in Subsection A.1.9.

The timetable generation algorithm requires the following **inputs**:

- *bus_line* or *bus_line_id*: The bus line where the generated timetables correspond to.

- *starting_datetime* and *ending_datetime* of timetables: The datetime period where the generated timetables correspond to.
- *starting_departure_datetime* and *ending_departure_datetime* of travel requests: Timetables are generated taking into consideration travel requests, with departure datetimes which correspond to the provided datetime period.

Apart from the inputs of the algorithm, the administrator of the system is responsible for making decisions regarding a set of parameters (*maximum bus capacity*, *minimum number of passengers in timetable*, and *average waiting time threshold*) which could affect the number of generated timetables, operating bus vehicles, average waiting time of passengers, and processing time.

The timetable generation algorithm includes the following **steps of execution**:

1. A connection is established between the Look Ahead component and the System Database, so as the list of bus stops corresponding to the provided bus line to be retrieved.
2. The list of travel request documents, corresponding to the selected bus line and with departure datetimes between the provided values, is retrieved by the Look Ahead component. In the following processing steps of timetable generation, the retrieved travel request documents are evaluated so as the minimum possible average waiting time of passengers to be ensured.
3. A request is sent from the Look Ahead to the Route Generator, so as the less time-consuming bus route connecting the bus stops of the bus line to be identified, while taking into consideration the current levels of traffic density. The response of the Route Generator includes details about starting and ending bus stops, covered distance, required travelling time, intermediate nodes, and edges which connect them.
4. Before starting the clustering of travel requests, some timetable documents should be generated, so as to be used as the initial clusters. For this reason, based on the response of the Route Generator and assuming that there is only one bus vehicle available, some initial timetables are generated by the Look Ahead, covering the whole datetime period between the provided starting and ending values. Initially, the lists of travel requests of these timetables are empty, since travel requests have not been partitioned yet, and the departure and arrival datetimes of the corresponding timetable entries are based exclusively on the travelling parameters of the followed route.
5. (Clustering of Travel Requests): Each one of the retrieved travel requests is corresponded to the timetable which produces the minimum waiting time for the passenger, which is calculated by comparing the departure datetime of the travel request with departure datetime of the corresponding timetable entry.
6. (Handling of Under-crowded Timetables): After the clustering step, there might be timetables where the total number of travel requests is lower than the value of the *minimum number of passengers in timetable* parameter. This could be usual during night hours, where transportation demand is not so high. These timetables are removed from the list of generated timetables, and each one of their travel requests is corresponded to one of the remaining timetables, taking into consideration the waiting time of the passenger.

7. (Handling of Over-crowded Timetables): In a similar way, there might be timetables where the number of current passengers is higher than the value of the *maximum bus capacity* parameter, which would indicate that each one of these timetables could not be served by one bus vehicle. For this reason, each one of these timetables should be divided into two timetables and the corresponding travel requests are partitioned, taking into consideration the waiting time of each passenger. The whole procedure is repeated until there is no timetable where the current number of passengers exceeds the maximum bus vehicle capacity.

8. (Adjusting Departure and Arrival Datetimes of Timetables): At this point of processing, the departure and arrival datetimes for each timetable entry are re-estimated, taking into consideration the departure datetimes of the corresponding travel requests and the required travelling time between bus stops. In each timetable and for each travel request, the ideal departure datetimes from all bus stops (not only the bus stop from where the passenger desires to depart) are estimated. Then, the ideal departure datetimes of the timetable, for each bus stop, correspond to the mean values of the ideal departure datetimes of the travel requests. Finally, starting from the initial bus stop and combining the ideal departure datetimes of each bus stop and the required travelling time between bus stops, included in the response of the Route Generator, the departure and arrival datetimes of the timetable entries are finalized.

9. (Handling of Timetables with Average Waiting Time Above Threshold): For each timetable, the average waiting time of passengers is calculated. If the average waiting time is higher than the value of the *average waiting time threshold* parameter, then the algorithm investigates the opportunity of splitting the timetable into two timetables. In this way, the travel requests of the initial timetable are partitioned, based on the waiting time of the passenger, and the departure and arrival datetimes of both timetables are re-estimated based on the departure datetimes of the partitioned travel requests. If the total number of travel requests in each timetable is higher than the minimum allowed number of passengers in timetable, and the average waiting time of each one of the produced timetables is lower than the corresponding waiting time of the initial timetable, then the new timetables are added to the list of generated timetables, and the initial one is removed. Otherwise, the initial timetable remains as it was before splitting. The whole procedure is repeated until there is no timetable with average waiting time above threshold, or no timetable partition could be performed.

10. The average waiting time of passengers in all timetables is calculated.

11. Steps 5 to 10 are repeated, as long as the average waiting time of passengers in all timetables is decreased.

12. The number of bus vehicles, required in order to serve the generated timetables, is calculated.

13. The generated timetable documents are stored to the corresponding collection of the System Database.

Taking into consideration the execution steps of the timetable generation algorithm, there is no doubt that the outputs and processing time of the provided solution are related to the adjustments

which are performed to the generated timetables, including re-estimation of timetable entries and timetable splitting, and could be affected by the parameters which are initialized by the administrator of the system. More precisely:

- *minimum_number_of_passengers_in_timetable*: The minimum number of passengers that could be served by a timetable. The timetable generation algorithm makes sure that there is no generated timetable where the total number of served travel requests is lower than the value of this parameter. For this reason, the timetable splitting procedure is also affected by this parameter, since the number of passengers in each timetable is compared with its value before investigating the possibility of splitting the timetable. As a result, lower values of this parameter could lead to lower numbers of passengers in timetables and additionally, to higher numbers of generated timetables and measurements of processing time.

- *average_waiting_time_threshold*: A time parameter used in order to set a limit to the timetable generation algorithm, regarding the adjustments which are performed to the generated timetables in order to reduce the average waiting time of passengers in each timetable, since as long as the average waiting time of passengers in a timetable is greater than the selected threshold, then the timetable generation algorithm will investigate the opportunity of making adjustments to the timetable focused on reducing the average waiting time of passengers. For this reason, lower values of this parameter could increase the number of required adjustments and as a result, the processing time could also be increased.

- *maximum_bus_capacity*: The maximum number of passengers that could be transported by a bus vehicle. The timetable generation algorithm ensures that there is no generated timetable where the number of current passengers exceeds the value of this parameter. Combined with the value of the *minimum number of passengers in timetable* parameter, they could lead to timetables with higher numbers of passengers and as a result, to fewer generated timetables and lower measurements of processing time. On the other hand, if the number of passengers in a timetable is too high or too low, there might be notable differences regarding the ideal departure datetimes of passengers. In this case, the number of timetable adjustments could also be increased, leading to higher measurements of processing time.

The timetable generation algorithm produces the following **outputs**:

- The generated timetable documents.
- The number of bus vehicles, required in order to serve the generated timetables.
- The average waiting time of passengers in all timetables.

The timetable generation algorithm could be described by the **pseudocode** of Section B.3.

Timetable Update

The timetable update algorithm requires the following **input**:

- *bus_line* or *bus_line_id*: Used in order to retrieve the timetable documents which should be updated.

The timetable update algorithm includes the following **steps of execution**:

1. A connection is established between the Look Ahead component and the System Database, so as the timetable documents which correspond to the provided bus line to be retrieved.
2. The retrieved timetable documents are processed, so as the corresponding travel request documents which are served by the timetables to be collected.
3. A request is sent from the Look Ahead to the Route Generator, so as the less time-consuming bus route connecting the bus stops of the bus line to be identified, while taking into consideration the current levels of traffic density. The response of the Route Generator includes details about starting and ending bus stops, covered distance, required travelling time, intermediate nodes, and edges which connect them.
4. The entries of the timetables (departure and arrival datetimes) are updated, taking into consideration the parameters of the recently identified route.
5. At this point of processing, the possibility of making adjustments to the existing timetables is investigated, in order to decrease the average waiting time of passengers which might be increased due to the recent changes in timetable entries. For this reason, the steps 5 to 12 of the timetable generation algorithm (link: Timetable Generation) are performed.
6. Finally, the Look Ahead connects to the System Database in order to update the timetable documents which have been modified, remove the documents which have been deleted or merged with other ones, and store the new documents which have been generated.

The timetable update algorithm produces the following **outputs**:

- The updated timetable documents, including the ones which have been added to or removed from the list of timetables.
- The new number of bus vehicles, required in order to serve the timetables of the provided bus line.
- The average waiting time of passengers in the timetables of the provided bus line.

3.5 System Administration

As illustrated in Figure 3.1, the administrator of the system is responsible for providing the OpenStreetMap files that contain geospatial data of selected geographical areas, and initializing the parameters which are important for the operation of the implemented components.

3.5.1 Parameters

System Database

- *mongodb_host*: The name of the host where the System Database (*MongoDB*) is running.
- *mongodb_port*: The port on which the System Database (*MongoDB*) is listening.

Travel Requests Simulator

- *travel_requests_simulator_timeout*: A parameter representing the time interval (in seconds) during which the Travel Requests Simulator process is waiting, before new travel requests are generated.
- *travel_requests_simulator_max_operation_timeout*: A parameter representing the time interval (in seconds) during which the Travel Requests Simulator process is running.
- *travel_requests_simulator_min_number_of_documents*: The minimum number of travel requests generated (each time) by the Travel Requests Simulator process.
- *travel_requests_simulator_max_number_of_documents*: The maximum number of travel requests generated (each time) by the Travel Requests Simulator process.
- *travel_requests_simulator_datetime_distribution_weights*: A list containing 24 integer values, corresponding to a 24-hour period, used by the Travel Requests Simulator as comparison values (weights) for the distribution of generated travel requests. Greater comparison values lead to more generated travel requests during the corresponding hourly period.

Traffic Data Simulator

- *traffic_data_simulator_timeout*: A parameter representing the time interval (in seconds) during which the Traffic Data Simulator process is waiting, before new traffic density values are generated for the stored edge documents.
- *traffic_data_simulator_max_operation_timeout*: A parameter representing the time interval (in seconds) during which the Traffic Data Simulator process is running.

Traffic Data Parser

- *traffic_data_parser_timeout*: A parameter representing the time interval (in seconds) during which the Traffic Data Parser process is waiting, before the traffic density values of stored edge documents are updated, based on the traffic jam events which are retrieved from the CityPulse Data Bus.
- *traffic_data_parser_max_operation_timeout*: A parameter representing the time interval (in seconds) during which the Traffic Data Parser process running.

Route Generator

- *route_generator_host*: The name of the host where the Route Generator is running.
- *route_generator_port*: The port on which the Route Generator is listening.
- *route_generator_request_timeout*: A parameter representing the time interval (in seconds) during which a client is waiting for a response from the Route Generator.

Look Ahead

- *minimum_number_of_passengers_in_timetable*: The minimum number of passengers that should be served by a timetable. The timetable generation algorithm makes sure that there is no generated timetable where the total number of served travel requests is lower than the value of this parameter.
- *average_waiting_time_threshold*: A time parameter (in seconds) used in order to set a limit to the timetable generation algorithm, regarding the adjustments which are performed to the generated timetables in order to reduce the average waiting time of passengers in each timetable. Being more specific, as long as the average waiting time of passengers in a timetable is greater than the selected threshold, then the timetable generation algorithm investigates the opportunity of updating the timetable entries or splitting the timetable, focused on reducing the average waiting time of the corresponding passengers.
- *maximum_bus_capacity*: The maximum number of passengers that could be transported by a bus vehicle. The timetable generation algorithm ensures that there is no generated timetable where the number of current passengers exceeds the value of this parameter.
- *look_ahead_timetables_generator_timeout*: A parameter representing the time interval (in seconds) during which the Look Ahead process is waiting, before the timetable generation algorithm is applied again.
- *look_ahead_timetables_generator_max_operation_timeout*: A parameter representing the time interval (in seconds) during which the timetable generation algorithm is applied by the Look Ahead process.
- *look_ahead_timetables_updater_timeout*: A parameter representing the time interval (in seconds) during which the Look Ahead process is waiting, before the timetable update algorithm is applied again.
- *look_ahead_timetables_updater_max_operation_timeout*: A parameter representing the time interval (in seconds) during which the timetable update algorithm is applied by the Look Ahead process.

Chapter 4

Evaluation

4.1 Experimental Evaluation

4.1.1 Testing Machine Specifications

A number of experiments were conducted in order to evaluate the components of the implemented system. The hardware and software specifications of the testing machine are illustrated in Table 4.1.

HARDWARE	
Model	`Dell Inspiron 3542`
CPU	`Intel(R) Core(TM) i3-4005U CPU @ 1.70GHz`
RAM	`8 GiB`
SOFTWARE	
Operating System	`Ubuntu 16.04.1 LTS (Xenial Xerus) 64-bit`
Python	`2.7.12`
MongoDB	`3.2.12`
Gunicorn	`19.6.0`
Node.js	`6.9.5`

Table 4.1: Hardware and software specifications of testing machine.

4.1.2 OpenStreetMap Data Parsing Evaluation

The first part of experimental evaluation was related to OpenStreetMap data parsing. More precisely, the performed tests were focused on evaluating the ability of the OpenStreetMap Parser to extract geospatial data from OpenStreetMap files and populate the corresponding collections of the System Database (Address, BusStop, Edge, Node, Point, and Way). The city of Uppsala was selected as area of operation and the properties of the input OpenStreetMap file, as well as the number of retrieved documents and corresponding processing times are illustrated in Table 4.2.

OpenStreetMap File Properties	
Link	*uppsala.osm*
Size on Disk	`53.9 MB`
Maximum Latitude	`59.9365175`
Minimum Latitude	`59.78544`
Maximum Longitude	`17.8277041`
Minimum Longitude	`17.4442835`
Number of Retrieved Documents	
Addresses	`38187`
Bus Stops	`536`
Edges	`38697`
Nodes	`6699`
Points	`239845`
Ways	`2702`
OpenStreetMap Parser Processing Times	
Extract Geospatial Data	`4.98 sec`
Populate Collections	`11.56 sec`

Table 4.2: Evaluation of OpenStreetMap data parsing.

4.1.3 Bus Line Generation and Route Identification Evaluation

Bus Stop	Coordinates	Distance (m)	Traveling Time (sec)	Total Routes
"Centralstationen"	(59.857218, 17.6484065)	645.37	53.63	1
"Stadshuset"	(59.8603359, 17.6429401)	797.48	65.24	2
"Skolgatan"	(59.8640214, 17.6339536)	1356.71	117.92	1
"Ekonomikum"	(59.8612517, 17.6203258)	1846.41	158.93	1
"Rickomberga"	(59.8553308, 17.6096497)	1164.22	97.98	2
"Oslogatan"	(59.8516822, 17.6034093)	138.71	12.80	2
"Reykjaviksgatan"	(59.8508683, 17.6015471)	188.12	17.36	2
"Ekebyhus"	(59.8505751, 17.599065)	401.08	37.02	2
"Sernanders väg"	(59.8514695, 17.5921197)	462.25	41.01	1
"Flogsta centrum"	(59.8520751, 17.5849535)	1906.26	118.78	2
"Sernanders väg"	(59.8514695, 17.5921197)	401.08	37.02	2
"Ekebyhus"	(59.8505751, 17.599065)	188.12	17.36	2
"Reykjaviksgatan"	(59.8508683, 17.6015471)	138.71	12.80	2
"Oslogatan"	(59.8516822, 17.6034093)	638.47	54.78	1
"Rickomberga"	(59.8553308, 17.6096497)	892.01	72.98	1
"Ekonomikum"	(59.8612517, 17.6203258)	951.36	77.99	2
"Skolgatan"	(59.8640214, 17.6339536)	733.94	60.05	1
"Stadshuset"	(59.8603359, 17.6429401)	725.90	60.22	2
"Centralstationen"	(59.857218, 17.6484065)	-	-	-

Table 4.3: Bus line generation and route identification evaluation.

The second part of experimental evaluation was carried out in two stages:

1. Bus Line Generation: A bus line was generated, using the Look Ahead component, connecting a set of randomly selected bus stops (*"Centralstationen"*, *"Stadshuset"*, *"Skolgatan"*, *"Ekonomikum"*, *"Rickomberga"*, *"Oslogatan"*, *"Reykjaviksgatan"*, *"Ekebyhus"*, *"Sernanders väg"*, and *"Flogsta centrum"*).

2. Route Identification: The Route Generator component was used in order to identify all the possible route connections between the bus stops of the generated bus line. In addition, the less time-consuming route was selected. Details about the selected route, such as coordinates (latitude, longitude) of bus stops, distance and travelling time measurements, as well as the total number of intermediate route connections are illustrated in Table 4.3.

Finally, even though they are not presented in the aforementioned table, the responses of Route Generator include also information about intermediate nodes, points, and edges which connect them.

Processing time measurements for the bus line generation and route identification operations are presented in Table 4.4. While generating a new bus line, all the possible route connections between intermediate bus stops should be identified by the Route Generator and stored as bus stop waypoints documents at the System Database. The first measurement (`27.35 sec`) was observed while performing all the steps of this operation. On the other hand, in the second measurement (`0.03 sec`) route identification had been performed in advance and bus stop waypoints were already stored.

Operation	Processing Time (sec)
`Bus Line Generation`	`(27.35) (0.03)`
`All Possible Routes Identification` (`entire bus line`)	10.67
`Less Time-Consuming Route Identification` (`entire bus line`)	1.46

Table 4.4: Bus line generation and route identification processing time measurements.

4.1.4 Timetable Generation Evaluation

The third part of experimental evaluation was carried out in two stages:

1. Travel Requests Simulation: Using the Travel Requests Simulator, multiple travel request documents were generated and stored to the related collection of the System Database, corresponding to passengers travelling between the bus stops of the generated bus line.

2. Timetable Generation: The timetables of the testing bus line were generated by the Look Ahead and stored to the corresponding collection of the System Database, based on the registered travel request documents and the parameters of the less time-consuming route, identified by the Route Generator.

As it has already been mentioned in Subsection 3.4.6, the outputs and performance of the Timetable Generation algorithm are affected by the number of registered travel request documents and some parameters (*maximum bus capacity*, *average waiting time threshold*, and *minimum number of passengers in timetable*), which are initialized by the administrator of the system. For this reason, multiple experiments were carried out while modifying the aforementioned parameters. The measurements of these experiments are illustrated in the next tables and figures, which include information about the following variables:

- *Passengers (total)*: The total number of passengers in all timetables, corresponding to the number of registered travel request documents.

- *Timetables*: The total number of generated timetables.
- *Vehicles*: The total number of bus vehicles, required in order to transport the passengers of the generated timetables.
- *Passengers (average)*: The average number of passengers in each timetable.
- *Waiting Time*: The average waiting time (in seconds) of passengers in all timetables.
- *Processing Time*: The processing time (in seconds) required in order to generate the timetables of the system and store them to the corresponding collection of the System Database.

Case 1: Modifying Minimum Number of Passengers in Timetable

- *maximum bus capacity*: **100**
- *minimum number of passengers in timetable*: **[10, 20, 30, 40, 50]** *(10%, 20%, 30%, 40%, and 50% of bus capacity)*
- *average waiting time threshold*: **0 sec**
- *registered travel requests*: **10000**

Passengers (minimum)	Timetables	Vehicles	Passengers (average)	Waiting Time (sec)	Processing Time (sec)
10	725	12	13	74.48	146.36
20	342	6	29	130.38	82.91
30	243	5	41	181.01	46.44
40	174	4	57	251.22	39.14
50	147	4	68	346.18	32.15

Table 4.5: Experimental measurements of the Timetable Generation algorithm, while modifying the minimum number of passengers in timetable.

Observations In this case, the *minimum number of passengers in timetable* parameter is modified while the *maximum bus capacity*, *average waiting time threshold*, and *number of registered travel requests* remain stable. The timetable generation algorithm makes sure that there is no generated timetable where the total number of travel requests is lower than the value of this parameter. For this reason, the timetable splitting procedure is also affected by this parameter, since the number of passengers in each timetable is compared with its value before investigating the possibility of splitting the timetable. As a result, as illustrated in Table 4.5 and Figures 4.1, 4.2, and 4.3, lower values for the *minimum number of passengers in timetable* parameter lead to an increase in the number of generated timetables, since the total number of passengers is not affected and the average number of passengers per timetable is decreased. Moreover, the number of required bus vehicles and processing time are also increased. On the other hand, the average waiting time of passengers is decreased.

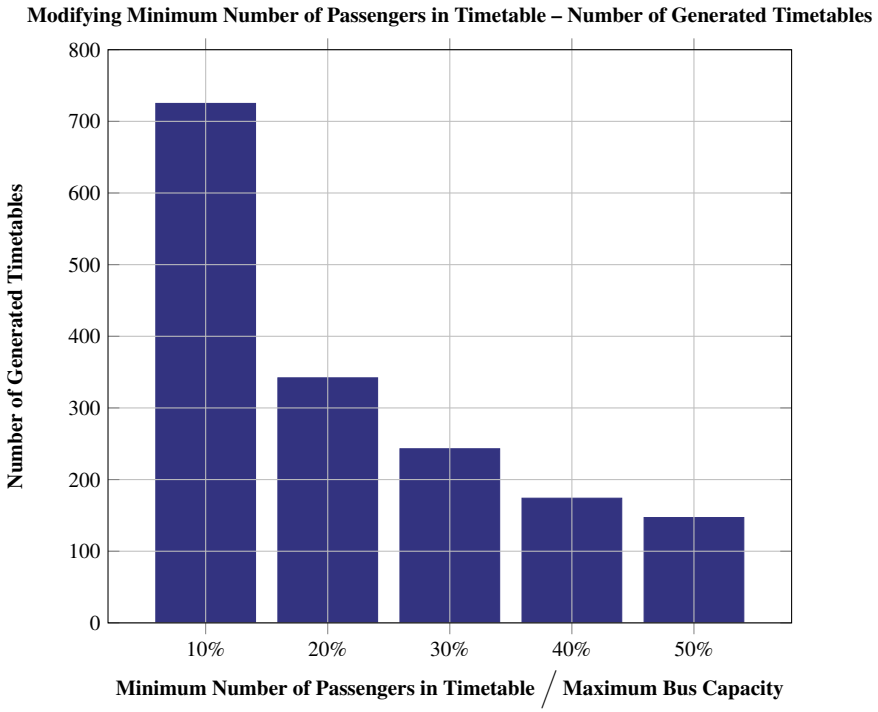

Figure 4.1: Experimental measurements of the Timetable Generation algorithm, regarding the number of generated timetables, while modifying the minimum number of passengers in timetable.

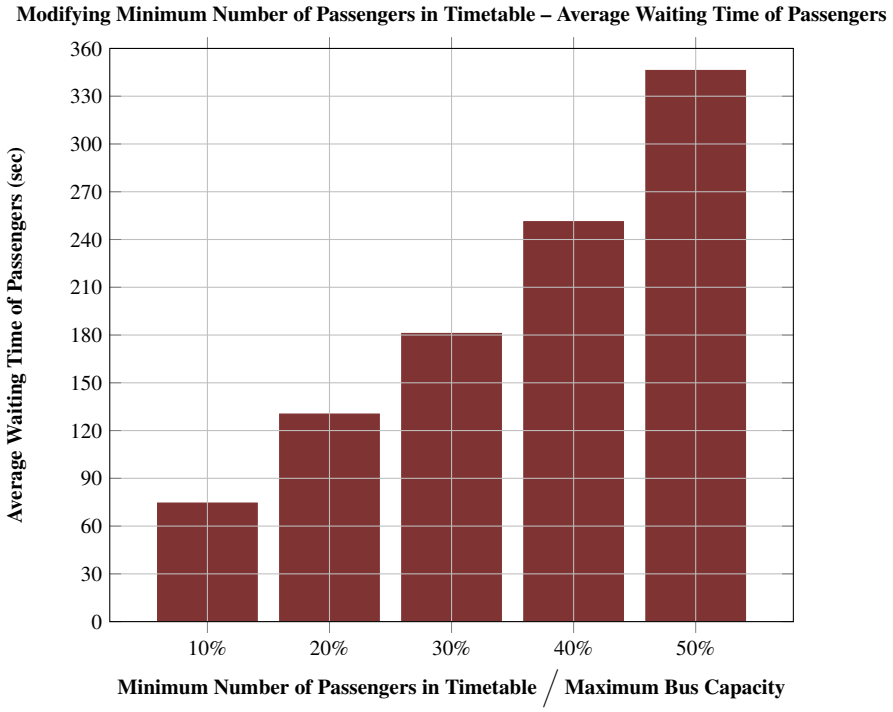

Figure 4.2: Experimental measurements of the Timetable Generation algorithm, regarding the average waiting time of passengers, while modifying the minimum number of passengers in timetable.

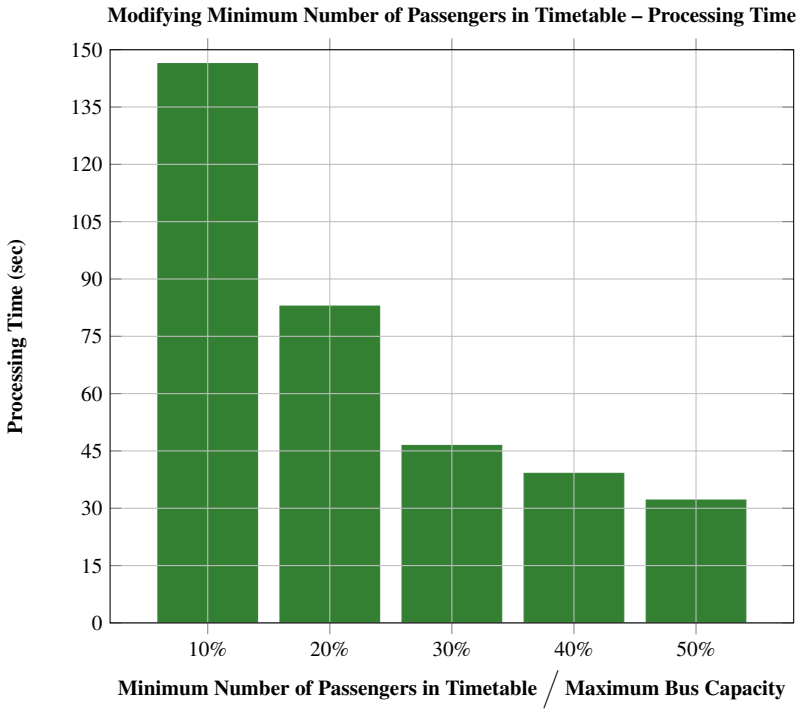

Figure 4.3: Experimental measurements of the Timetable Generation algorithm, regarding processing time, while modifying the minimum number of passengers in timetable.

Case 2: Modifying Maximum Bus Capacity and Minimum Number of Passengers in Timetable

- *maximum bus capacity*: **[40, 80, 120, 160, 200]**
- *minimum number of passengers in timetable*: **[20, 40, 60, 80, 100]** *(50% of bus capacity)*
- *average waiting time threshold*: **0 sec**
- *registered travel requests*: **10000**

Bus Capacity	Timetables	Vehicles	Passengers (average)	Waiting Time (sec)	Processing Time (sec)
40	349	6	28	235.60	59.09
80	175	4	57	342.36	42.69
120	121	3	82	350.30	24.99
160	97	3	103	360.95	23.30
200	64	3	156	542.32	59.35

Table 4.6: Experimental measurements of the Timetable Generation algorithm, while modifying the maximum bus capacity and minimum number of passengers in timetable.

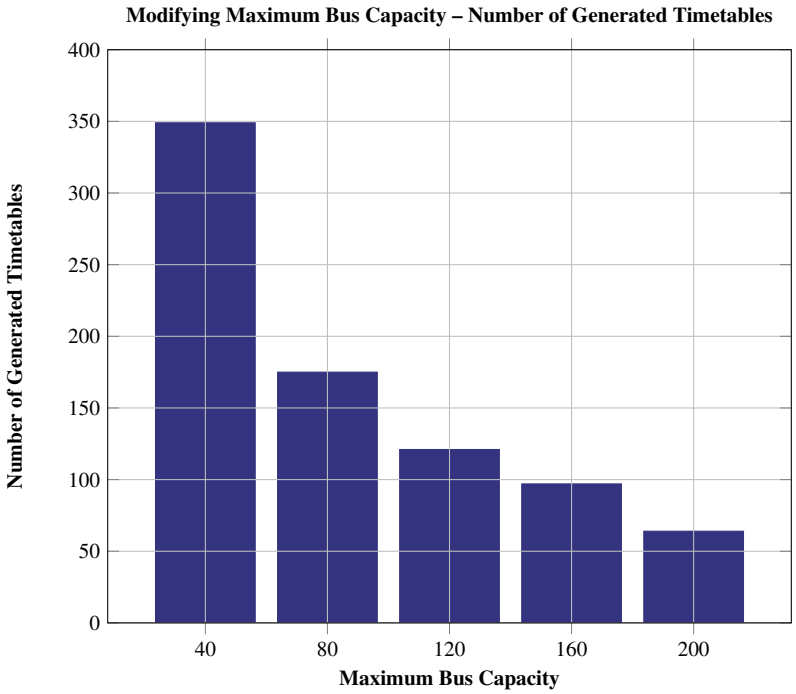

Figure 4.4: Experimental measurements of the Timetable Generation algorithm, regarding the number of generated timetables, while modifying the maximum bus capacity and minimum number of passengers in timetable.

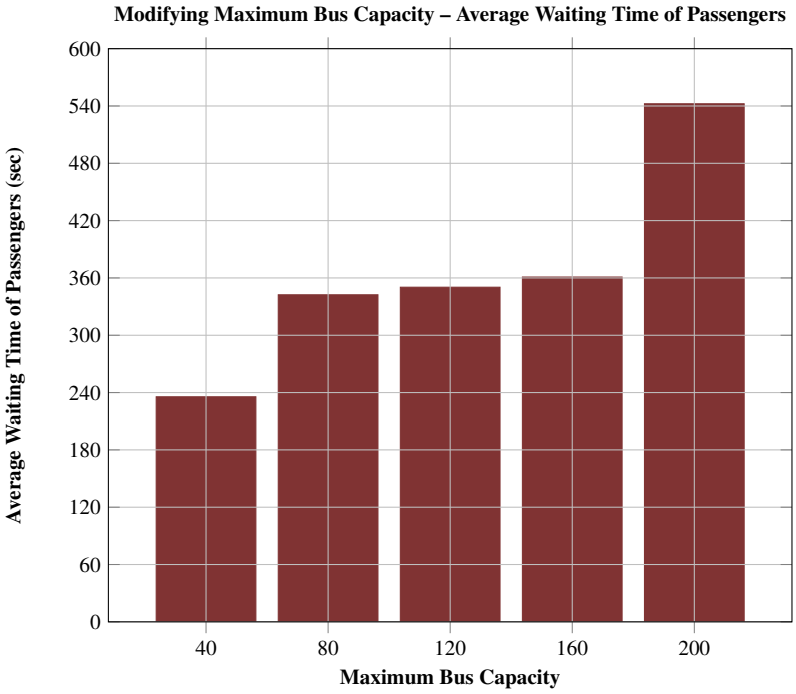

Figure 4.5: Experimental measurements of the Timetable Generation algorithm, regarding the average waiting time of passengers, while modifying the maximum bus capacity and minimum number of passengers in timetable.

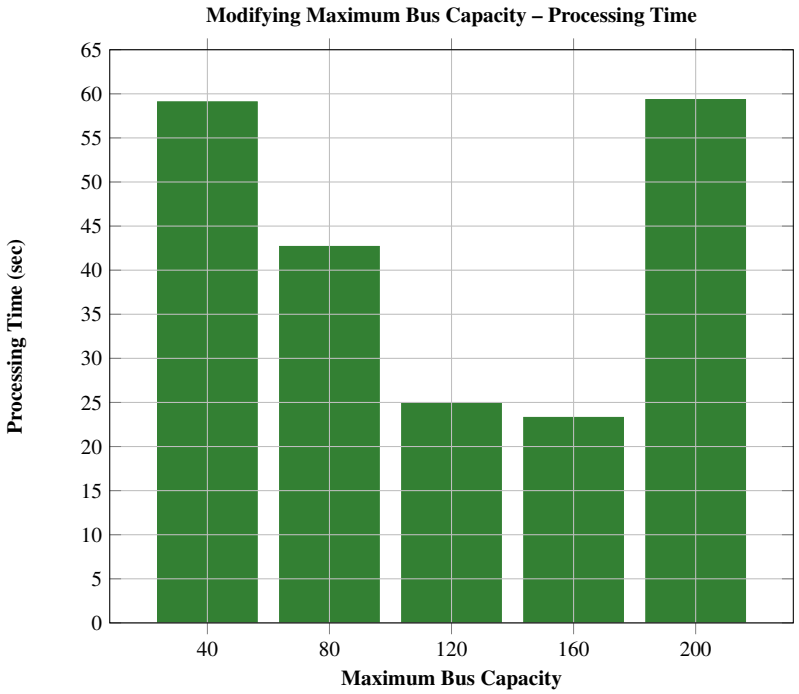

Figure 4.6: Experimental measurements of the Timetable Generation algorithm, regarding processing time, while modifying the maximum bus capacity and minimum number of passengers in timetable.

Observations In this case, the *maximum bus capacity* and *minimum number of passengers in timetable* parameters are modified, while the *average waiting time threshold* and *number of registered travel requests* remain stable. The timetable generation algorithm ensures that there is no generated timetable where the number of current passengers exceeds the value of the *maximum bus capacity* parameter. In addition, as illustrated in Table 4.6 and Figures 4.4, 4.5, and 4.6, combined with the value of the *minimum number of passengers in timetable* parameter, they could lead to timetables with higher numbers of passengers and as a result, to fewer generated timetables. On the other hand, the average waiting time could also be increased and in cases where the average number of passengers per timetable is too low or too high (e.g., maximum bus capacity equal to 40 or 200), there might be notable differences regarding the ideal departure datetime of passengers and as a consequence, the number of required timetable adjustments could also be increased leading to higher measurements of processing time.

Case 3: Modifying Average Waiting Time Threshold

- *maximum bus capacity*: **100**
- *minimum number of passengers in timetable*: **10**
- *average waiting time threshold*: **[90, 120, 150, 180, 210] sec**
- *registered travel requests*: **10000**

Observations In this case, the *average waiting time threshold* parameter is modified while the *maximum bus capacity*, *minimum number of passengers in timetable* and *number of registered travel requests* remain constant. The administrator of the system is offered the opportunity

to set a limit to the timetable generation algorithm, regarding the adjustments which are performed to the generated timetables in order to reduce the average waiting time of passengers. More precisely, as long as the average waiting time of passengers in a timetable is greater than the provided threshold, then the timetable generation algorithm investigates the possibility of making adjustments to the timetable (re-estimation of timetable entries or timetable splitting), focused on reducing the average waiting time of passengers. For this reason, lower values of this parameter could increase the number of required adjustments and as illustrated in Table 4.7 and Figures 4.7, 4.8, and 4.9, the number of generated timetables, required bus vehicles, and corresponding processing time could also be increased. On the other hand, the average number of passengers per timetable and the average waiting time of passengers could be decreased.

Waiting Time Threshold (sec)	Timetables	Vehicles	Passengers (average)	Waiting Time (sec)	Processing Time (sec)
90	502	9	19	91.56	176.88
120	413	8	24	108.96	116.34
150	296	7	33	149.62	50.12
180	196	5	51	222.58	35.73
210	171	4	58	255.46	32.04

Table 4.7: Experimental measurements of the Timetable Generation algorithm, while modifying the average waiting time threshold.

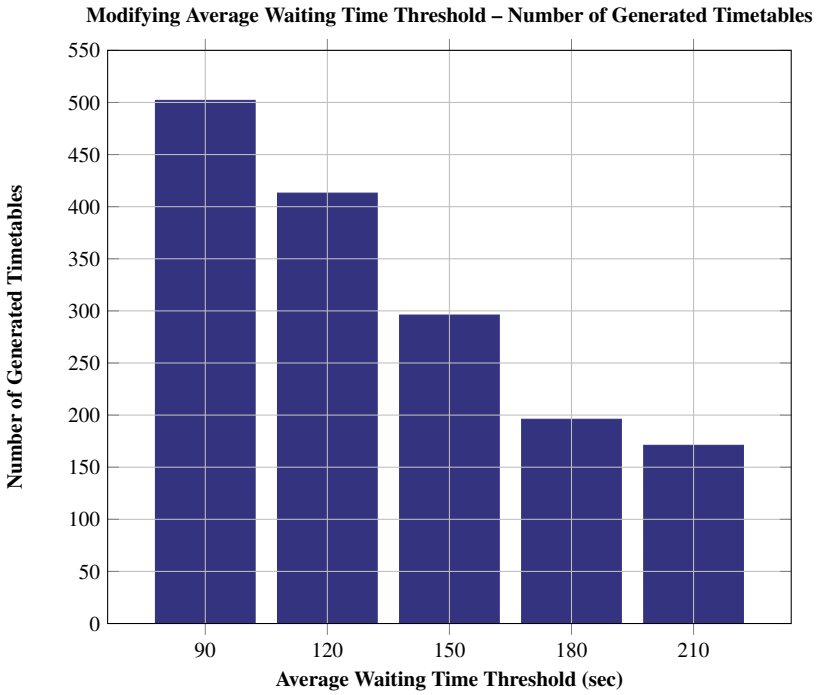

Figure 4.7: Experimental measurements of the Timetable Generation algorithm, regarding the number of generated timetables, while modifying the average waiting time threshold.

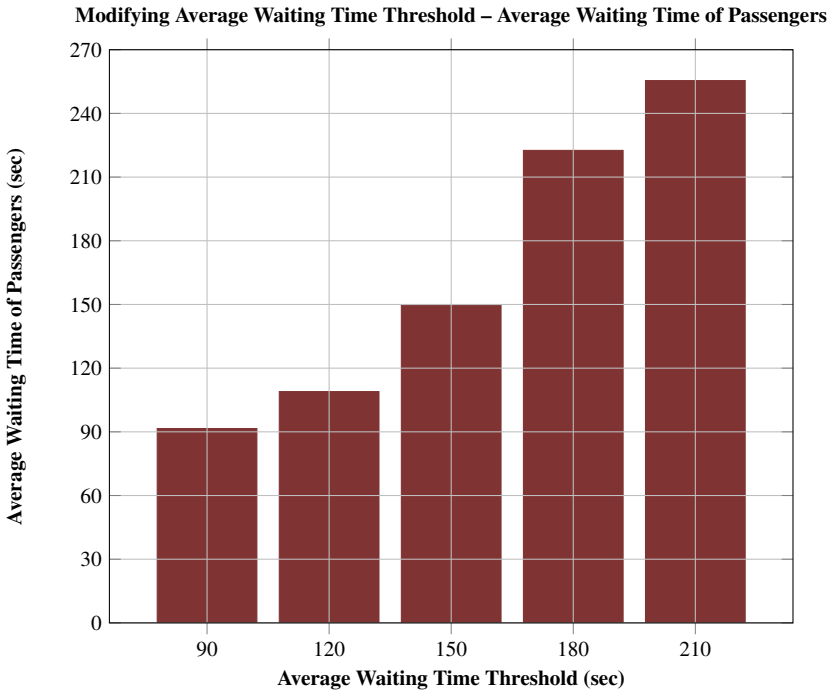

Figure 4.8: Experimental measurements of the Timetable Generation algorithm, regarding the average waiting time of passengers, while modifying the average waiting time threshold.

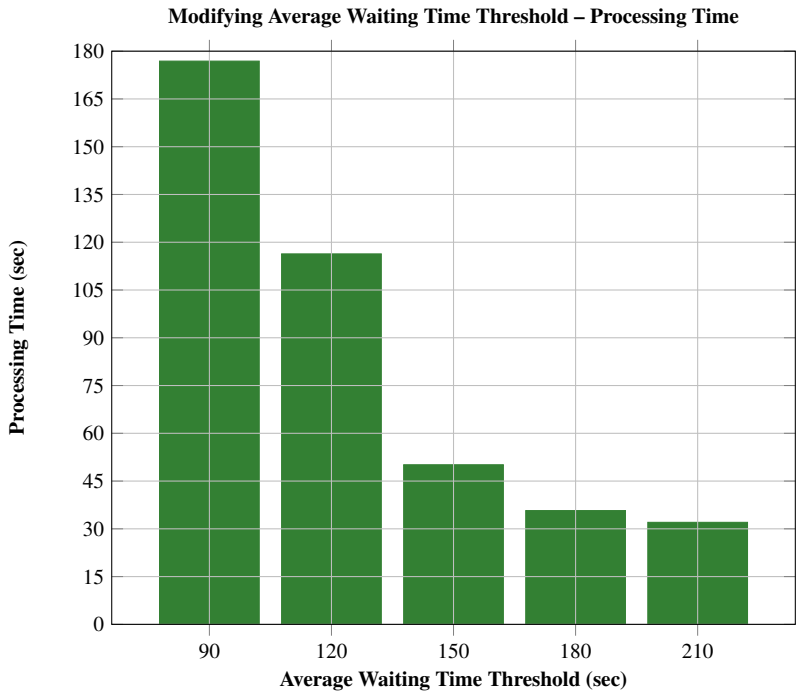

Figure 4.9: Experimental measurements of the Timetable Generation algorithm, regarding processing time, while modifying the average waiting time threshold.

Case 4: Modifying Registered Travel Requests

- *maximum bus capacity*: **100**
- *minimum number of passengers in timetable*: **10**
- *average waiting time threshold*: **0 sec**
- *registered travel requests*: **[2000, 4000, 6000, 8000, 10000]**

Passengers (total)	Timetables	Vehicles	Passengers (average)	Waiting Time (sec)	Processing Time (sec)
2000	138	3	14	180.99	26.13
4000	285	6	14	112.42	79.45
6000	439	8	13	87.33	104.07
8000	576	10	13	84.20	110.77
10000	725	12	13	74.48	146.36

Table 4.8: Experimental measurements of the Timetable Generation algorithm, while modifying the number of registered travel requests.

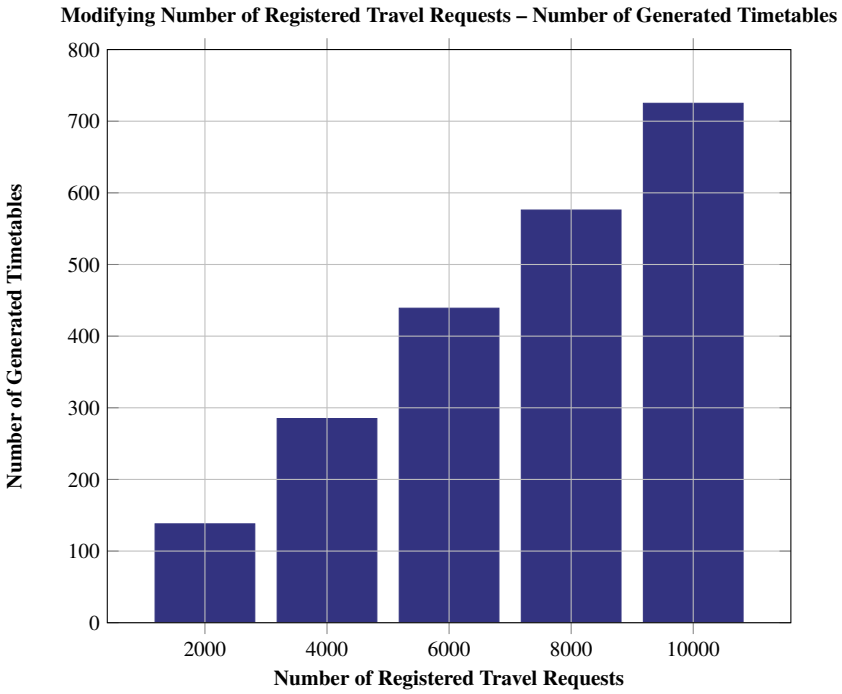

Figure 4.10: Experimental measurements of the Timetable Generation algorithm, regarding the number of generated timetables, while modifying the number of registered travel requests.

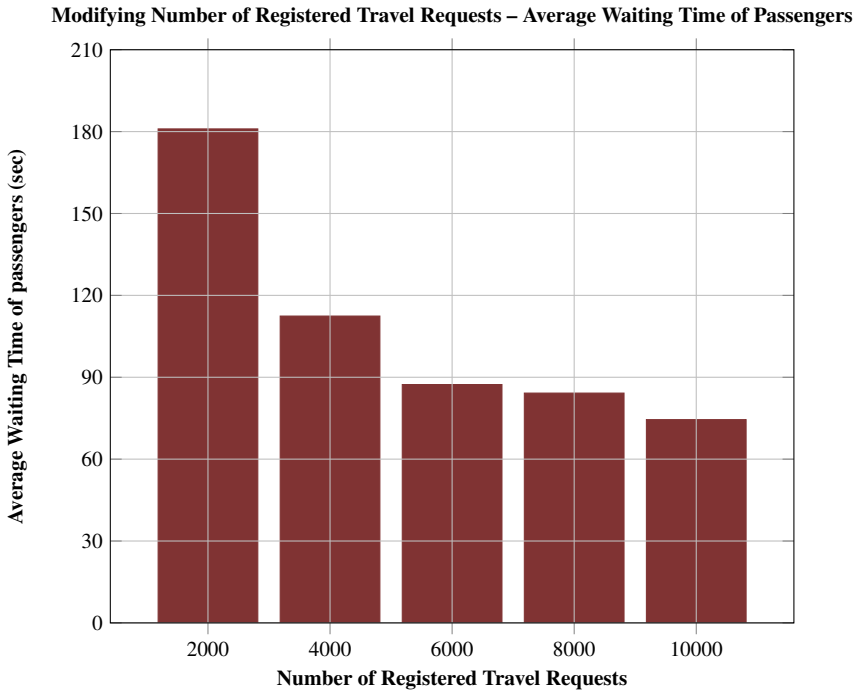

Figure 4.11: Experimental measurements of the Timetable Generation algorithm, regarding the average waiting time of passengers, while modifying the number of registered travel requests.

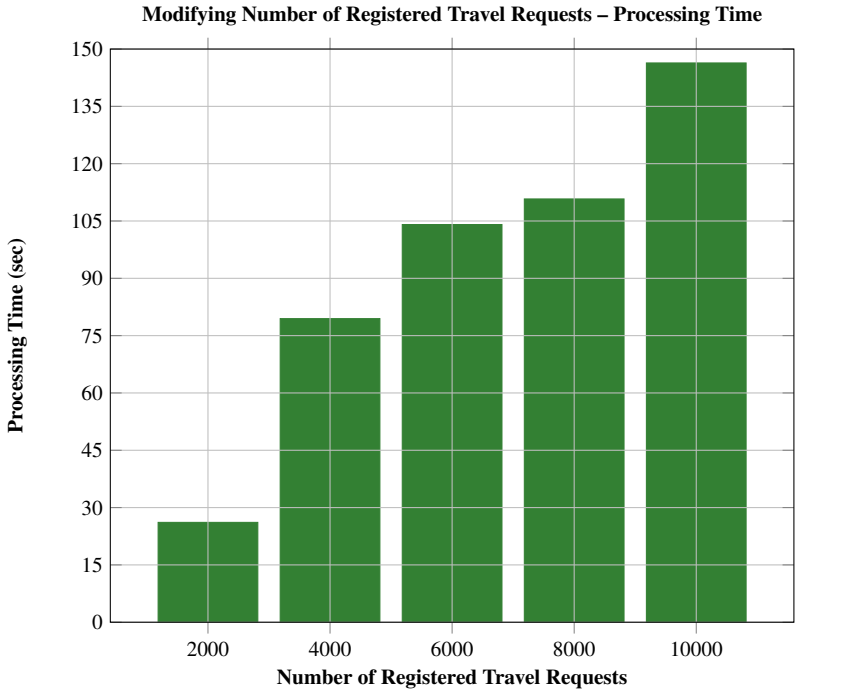

Figure 4.12: Experimental measurements of the Timetable Generation algorithm, regarding processing time, while modifying the number of registered travel requests.

Observations In this case, the *number of registered travel requests* parameter is modified while the *maximum bus capacity*, *minimum number of passengers in timetable*, and *average*

waiting time threshold remain stable. As illustrated in Table 4.8 and Figures 4.10, 4.11, and 4.12, more travel requests lead to increased numbers of generated timetables and required bus vehicles. In addition, processing time measurements are increased as well. Finally, it seems that the average waiting time of passengers is decreased, which could be explained by the fact that the departure datetimes of travel requests follow a uniform distribution and higher numbers of travel requests lead to timetables with improved cohesion, as far as departure datetime distribution is concerned.

4.1.5 Traffic Data Handling and Timetable Update Evaluation

The final part of experimental evaluation was carried out in three stages:

1. Traffic Data Simulation: A set of traffic density values was generated, utilizing the Traffic Data Simulator component, corresponding to the edge documents which connect the bus stops of the generated bus line.

2. Route Identification: A new route was identified by the Route Generator component, taking into consideration the updated traffic density values.

3. Timetable Update: Based on the parameters of the new route, the Look Ahead component generated new timetable documents or updated the corresponding entries of the already existing ones.

Modifying Traffic Density – Route Identification

Traffic Density	Covered Distance (km)	Traveling Time (min)	Average Speed (km/h)
0%	13.57	18.56	43.87
10% - 30%	13.57	20.31	40.10
30% - 50%	13.57	23.25	35.03
50% - 70%	13.57	29.20	27.89
70% - 90%	13.57	37.76	21.57

Table 4.9: Experimental measurements of route identification, while modifying the levels of traffic density.

Observations As illustrated in Table 4.9 and Figures 4.13 and 4.14, travelling time and average speed of bus vehicles are affected by changes in traffic density. More precisely, higher values of traffic density lead to decreased average speed measurements and as a consequence, travelling time is increased.

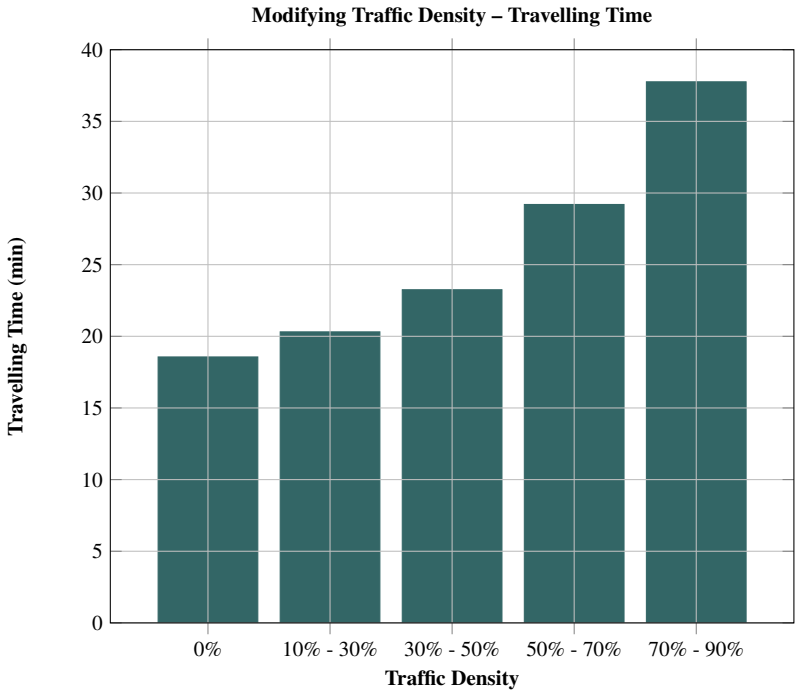

Figure 4.13: Experimental measurements of route identification, regarding travelling time, while modifying the levels of traffic density.

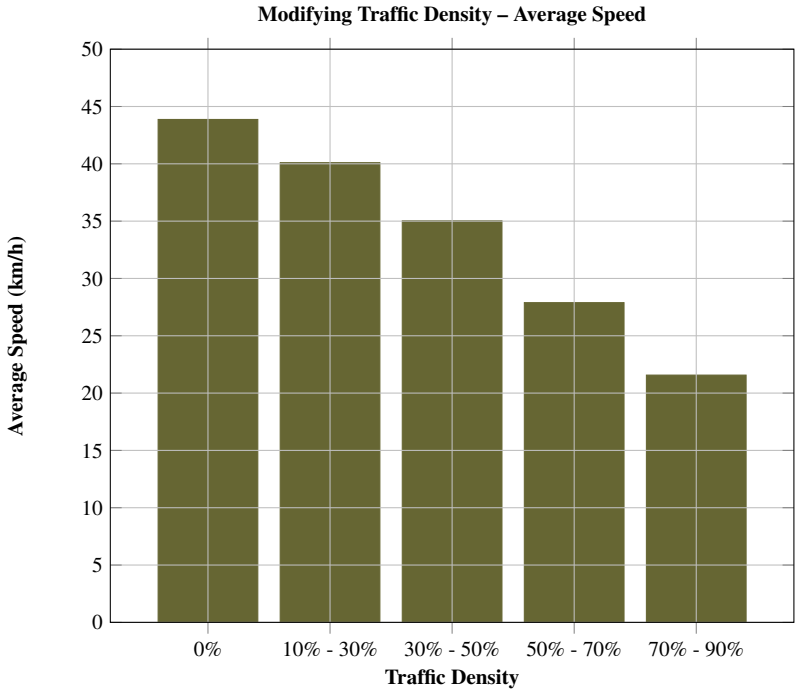

Figure 4.14: Experimental measurements of route identification, regarding average speed, while modifying the levels of traffic density.

Case 1: Modifying Traffic Density – Minimum Number of Passengers in Timetable – 10% of Bus Capacity

- *maximum bus capacity*: **100**
- *minimum number of passengers in timetable*: **10** (*10% of bus capacity*)
- *average waiting time threshold*: **0 sec**
- *registered travel requests*: **10000**

Traffic Density	Timetables	Vehicles	Passengers (average)	Waiting Time (sec)	Processing Time (sec)
0%	725	12	13	74.48	146.36
10% - 30%	733	14	13	81.62	136.73
30% - 50%	731	16	13	91.12	131.47
50% - 70%	720	18	13	96.87	138.46
70% - 90%	713	24	14	113.55	145.59

Table 4.10: Experimental measurements of the Timetable Generation algorithm, while modifying the levels of traffic density, with minimum number of passengers in timetable corresponding to 10% of bus capacity.

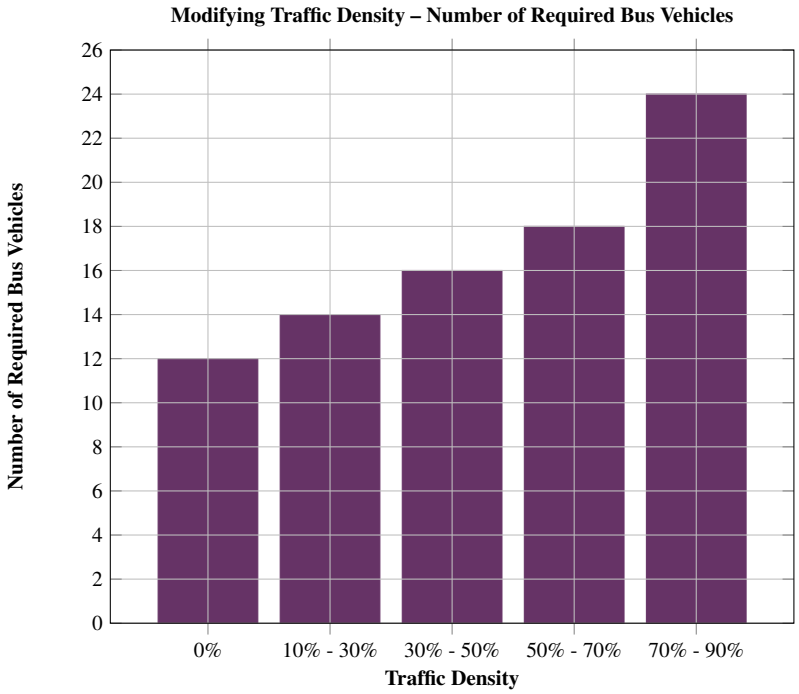

Figure 4.15: Experimental measurements of the Timetable Generation algorithm, regarding the number of required bus vehicles, while modifying the levels of traffic density and with minimum number of passengers in timetable corresponding to 10% of bus capacity.

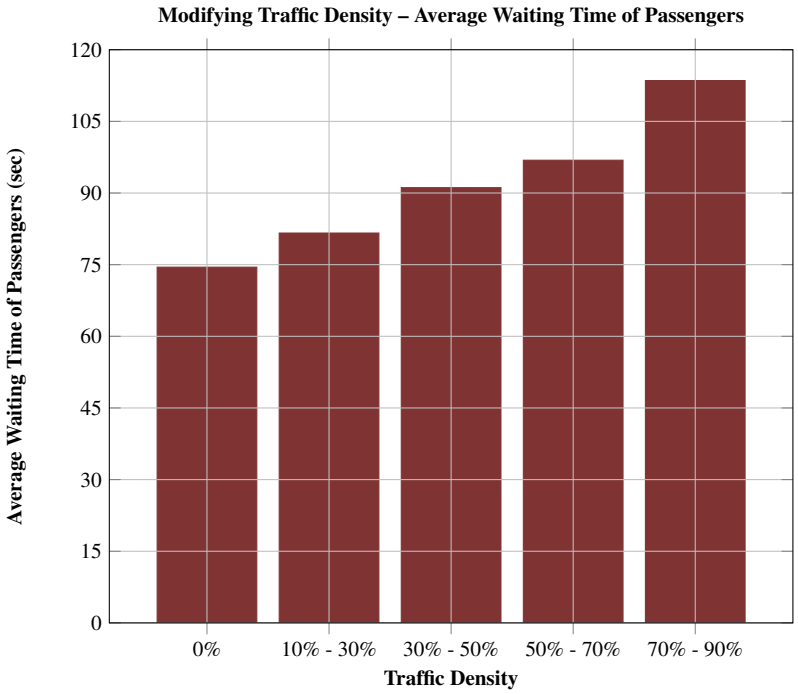

Figure 4.16: Experimental measurements of the Timetable Generation algorithm, regarding the average waiting of passengers, while modifying the levels of traffic density and with minimum number of passengers in timetable corresponding to 10% of bus capacity.

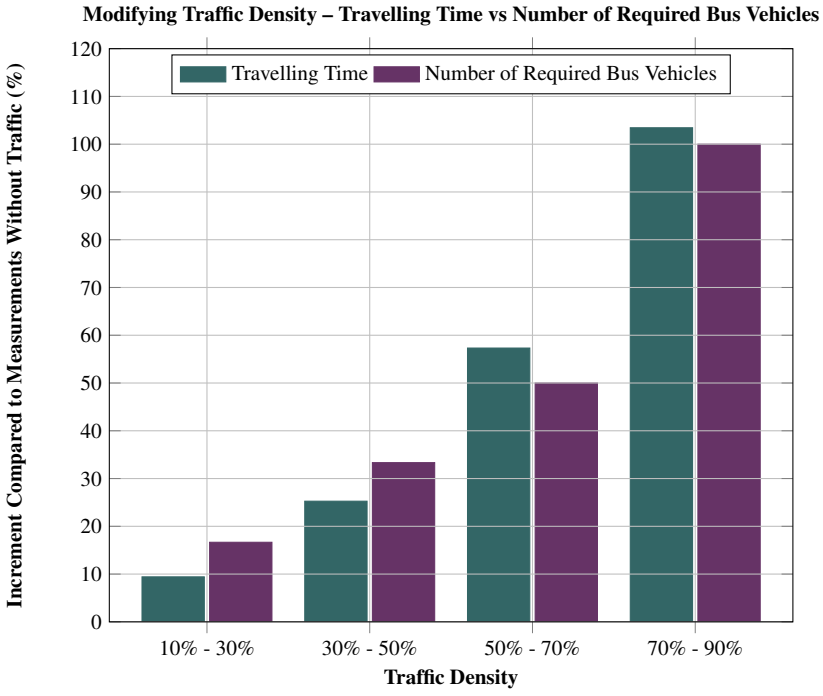

Figure 4.17: Experimental measurements of the Timetable Generation algorithm, illustrating a comparison between travelling time and number of required bus vehicles, while modifying the levels of traffic density and with minimum number of passengers in timetable corresponding to 10% of bus capacity.

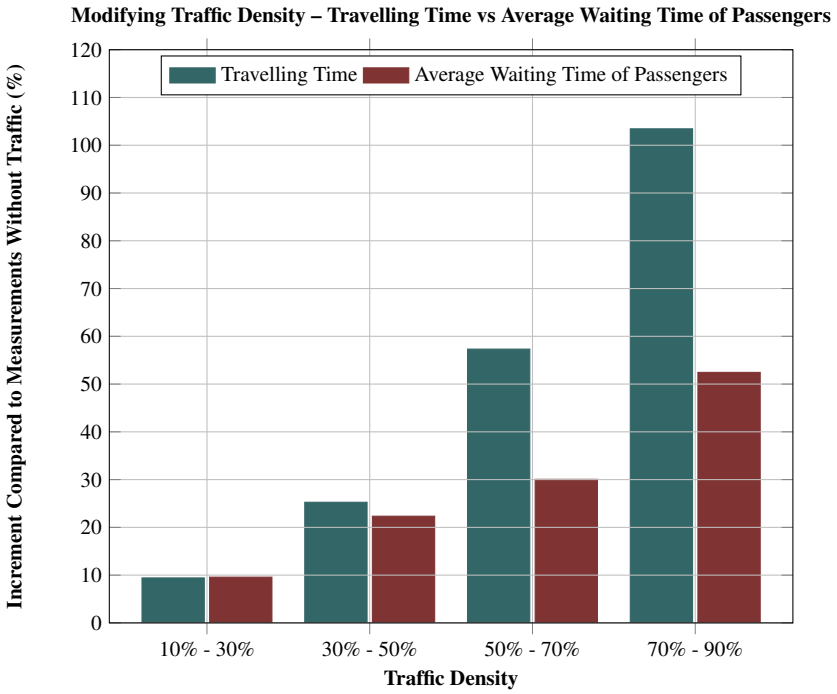

Figure 4.18: Experimental measurements of the Timetable Generation algorithm, illustrating a comparison between travelling time and average waiting of passengers, while modifying the levels of traffic density and with minimum number of passengers in timetable corresponding to 10% of bus capacity.

Observations In this case, traffic density values are modified while selecting a low value for the *minimum number of passengers in timetable* parameter, corresponding to 10% of maximum bus capacity, and keeping the *maximum bus capacity*, *average waiting time threshold*, and *number of registered travel requests* parameters constant. As illustrated in Table 4.10 and Figures 4.15, 4.16, 4.17, 4.18, the number of generated timetables, average number of passengers per timetable, as well as the processing time measurements are not drastically affected by changes in traffic. On the other hand, higher values of traffic density lead to increased number of required bus vehicles and average waiting time measurements. In addition, there is no doubt that the number of required vehicles is more affected by the modified levels of traffic density, compared to the average waiting time of passengers. This observation could be explained by the low value of the *minimum number of passengers in timetable* parameter, which affects the timetable generation algorithm in order to achieve the best possible result regarding the average waiting time of passengers. As a result, more bus vehicles are required so as to avoid a notable increment in the average waiting time of passengers, due to increased travelling time.

Case 2: Modifying Traffic Density – Minimum Number of Passengers in Timetable – 50% of Bus Capacity

- *maximum bus capacity*: **100**
- *minimum number of passengers in timetable*: **50** (*50% of bus capacity*)
- *average waiting time threshold*: **0 sec**
- *registered travel requests*: **10000**

Traffic Density	Timetables	Vehicles	Passengers (average)	Waiting Time (sec)	Processing Time (sec)
0%	147	4	68	346.18	32.15
10% - 30%	150	4	66	410.53	26.42
30% - 50%	144	4	69	450.56	25.33
50% - 70%	139	5	71	505.34	23.58
70% - 90%	151	6	66	560.44	25.48

Table 4.11: Experimental measurements of the Timetable Generation algorithm, while modifying the levels of traffic density, with minimum number of passengers in timetable corresponding to 50% of bus capacity.

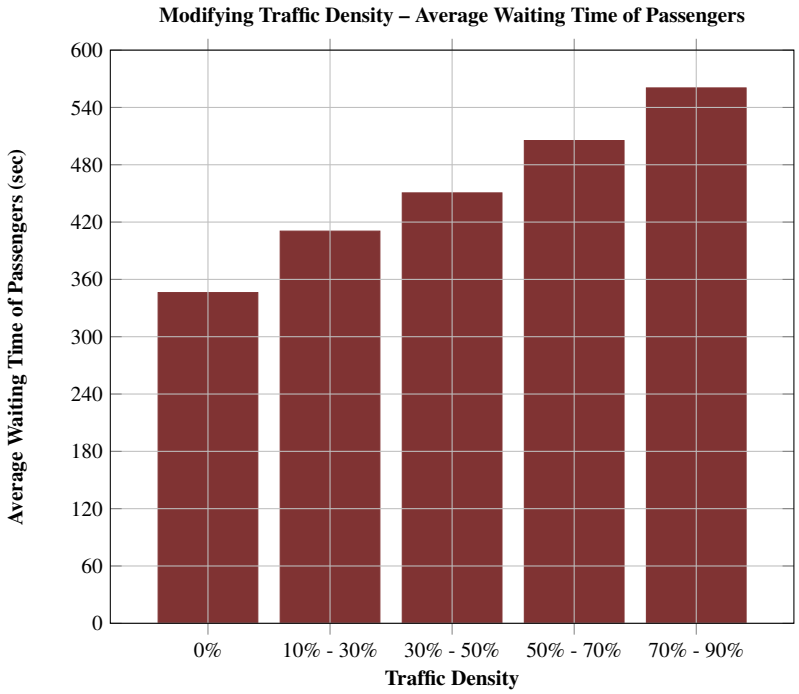

Figure 4.19: Experimental measurements of the Timetable Generation algorithm, regarding the average waiting of passengers, while modifying the levels of traffic density and with minimum number of passengers in timetable corresponding to 50% of bus capacity.

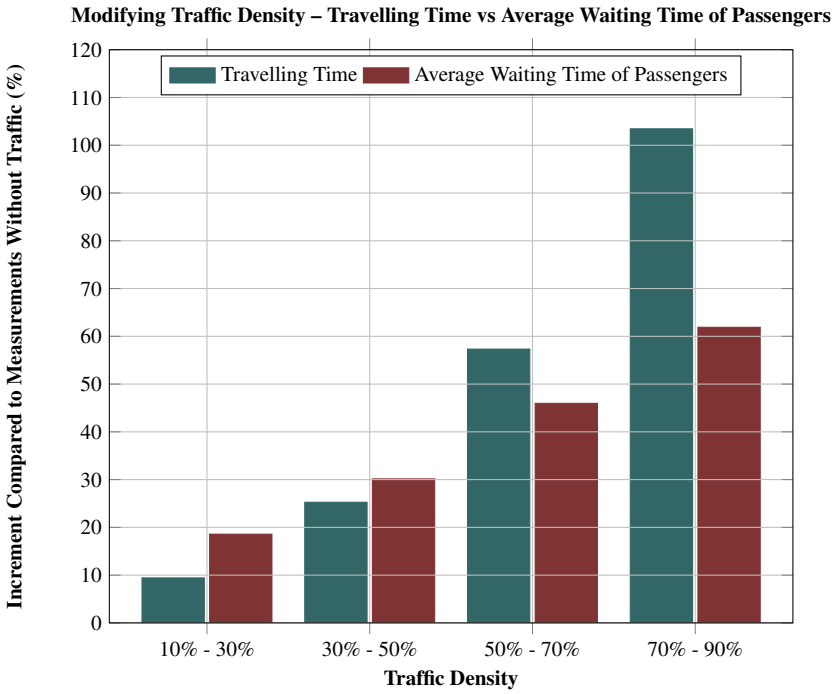

Figure 4.20: Experimental measurements of the Timetable Generation algorithm, illustrating a comparison between travelling time and average waiting of passengers, while modifying the levels of traffic density and with minimum number of passengers in timetable corresponding to 50% of bus capacity.

Observations In this case, traffic density values are modified while selecting a high value for the *minimum number of passengers in timetable* parameter, corresponding to 50% of maximum bus capacity, and keeping the *maximum bus capacity*, *average waiting time threshold*, and *number of registered travel requests* parameters constant. As illustrated in Table 4.11 and Figures 4.19, 4.20, the number of generated timetables, required bus vehicles, the average number of passengers per timetable, as well as the processing time measurements are not drastically affected by changes in traffic density. On the other hand, higher values of traffic density lead to increased measurements regarding the average waiting time of passengers, despite the fact that this increase is not so high compared to the increment in travelling time.

4.2 Comparison with Similar Projects

Mobile Network Assisted Driving (MoNAD) is a project of *Ericsson Research*, implemented in collaboration with *Uppsala University*, which is also focused on developing a bus management system capable of generating transportation schedules. *Bus Scheduling including Dynamic Events* could be considered as an extension or alternative of *MoNAD*, introducing a new algorithm for travel request evaluation and timetable generation, while providing features such as real-time traffic flow detection and dynamic route identification. Experimental evaluation proved that the introduced algorithm is capable of demonstrating improved measurements concerning the average waiting time of passengers, compared to *MoNAD*, while decreasing considerably processing time. An overview regarding the main features of the two projects is illustrated in Table 4.12.

Project	Bus Scheduling including Dynamic Events	Mobile Network Assisted Driving
Timetable Generation	Timetable generation is based on dynamic clustering techniques. Each timetable could be considered as a cluster, since it contains multiple travel requests which are partitioned based on their departure datetime values and affect the corresponding timetable entries. The implemented algorithm offers the opportunity to the administrator of the system to make decisions regarding a set of parameters (*maximum bus capacity*, *minimum number of passengers in timetable*, and *average waiting time threshold*), which affect outputs of the algorithm such as the generated timetables, bus vehicles required in order to transport the passengers, waiting and processing time.	Timetable generation is based on a genetic algorithm which combines operations such as mutation and crossover. The algorithm is focused on providing the best possible fitness value (waiting time of passengers), while allowing the administrator of the system to select the number of generations and individuals, which affect the fitness value and processing time. Compared to the current project, there is no support for changing the number of generated timetables or the number of passengers per timetable, and vehicle resources are considered unlimited.
Route Identification	The Route Generator component offers the opportunity of identifying the less time-consuming route between a list of provided bus stops, utilizing the *A** search algorithm. In addition, the current project is also capable of identifying all the possible routes and intermediate waypoints, introducing a variation of the *Breadth-first* search algorithm.	Taking advantage of the *A** search algorithm, *MoNAD* is able to identify the less time-consuming route connecting a list of bus stops. Compared to the current project, there is no support for identifying all the possible routes between the provided bus stops.
Traffic Data Handling	The implemented system is capable of receiving and processing real-time traffic data, from the *CityPulse Data Bus*, while making adjustments to the predefined routes of bus vehicles in order to identify the less time-consuming ones, based on the current levels of traffic density, and update the corresponding timetable entries.	–
Front-end Applications	–	Two *Android* applications addressed to passengers and drivers respectively.

Table 4.12: *Bus Scheduling including Dynamic Events* vs *Mobile Network Assisted Driving*.

The experimental measurements of the timetable generation algorithm of *Mobile Network Assisted Driving* are presented Table 4.13. In a genetic algorithm, a population of candidate solutions, called *"individuals"*, to an optimization problem is evolved toward better solutions. Each candidate solution has a set of properties which can be mutated and altered. The evolution usually starts from a population of randomly generated individuals, and is an iterative process, with the population in each iteration called as *"generation"*. In each generation, the fitness of every individual in the population is evaluated; the fitness is usually the value of the objective function in the optimization problem being solved. The more fit individuals are stochastically selected

from the current population, and the genome of each individual is modified (recombined and possibly randomly mutated) to form a new generation. The new generation of candidate solutions is then used in the next iteration of the algorithm. The execution of the algorithm is usually terminated when either a maximum number of generations has been produced, or a satisfactory fitness level has been reached for the population [33].

Number of Generations	Number of Individuals	Average Waiting Time of Passengers (sec)	Processing Time (min)
10	10	520	2
50	100	256	52
50	200	209	105
50	400	161	205
50	800	156	390

Table 4.13: Experimental measurements of the timetable generation algorithm of *Mobile Network Assisted Driving* (according to the final report of the project).

As illustrated in the table, while the numbers of generations and individuals are increased an improved fitness value is observed, corresponding to lower average waiting time for passengers. On the other hand, the required processing time of the algorithm is remarkably increased. Finally, taking into consideration corresponding measurements of the timetable generation algorithm of *Bus Scheduling including Dynamic Events* (Lowest Average Waiting Time of Passengers: `74.48 sec` – Corresponding Processing Time: `146.36 sec`), there is no doubt that the dynamic clustering approach which is introduced in this project is more efficient regarding the average waiting time of passengers and significantly faster.

Chapter 5

Conclusions and Future Work

Bus Scheduling including Dynamic Events is a project focused on developing a realistic simulation of a transportation management system, capable of identifying connections among the road network of operation areas, creating bus lines consisting of multiple connected bus stops, simulating travel requests registered by potential passengers, and generating routes and timetables for bus vehicles, while taking into consideration factors which could affect the predefined schedule, such as unpredictable events (e.g., traffic accidents) or dynamic levels of traffic density.

The implemented system introduces a reasoning mechanism capable of evaluating travel requests and generating timetables, utilizing dynamic clustering techniques, while offering the opportunity to its administrator to make decisions regarding the number of generated timetables, required vehicles, passengers per timetable, waiting time of passengers and processing time. In addition, routes are identified dynamically, taking into consideration real-time traffic data, in order to select the less time-consuming connections between bus stops and make adjustments to the corresponding timetables.

Experimental evaluation (Section 4.1) proved that the introduced timetable generation algorithm is capable of demonstrating considerable efficiency while evaluating registered travel requests, generating timetables with the average waiting time of passengers equal to `74.48` seconds. Moreover, the dynamic clustering approach of the new algorithm offered a remarkable improvement regarding processing time, compared to the corresponding measurements of similar algorithms (Section 4.2), being able to evaluate `10000` travel requests and generate `725` timetables in `146.36` seconds.

On the other hand, there is no doubt that there are still opportunities for further enhancements regarding:

- **OpenStreetMap Data**: *OpenStreetMap (OSM)* is a collaborative project towards the creation of a free and editable map of the world. In this project, *OpenStreetMap* is used in order to provide information regarding the road network of operation areas. The main objection towards the selection of *OpenStreetMap* is the lack of restrictions concerning usage rights, which offers the opportunity of testing the implemented system without any kind of limitation. On the other hand, experimental evaluation indicated that there might be cases where *OpenStreetMap* datasets might not be complete (e.g., lack of addresses, bus stops without names or coordinates, or absence of speed limits in roads). For this reason, adding support for additional/alternative data sources, regarding the road network

of operation areas, could be valuable improvement for the implemented system.

- **Traffic Data Sources**: The implemented system is capable of retrieving real-time traffic data from the *CityPulse Data Bus*, collected by fixed sensors in the city infrastructure (e.g., streets, public buildings, or utility systems) or in personal and business properties (private vehicles or buildings). On the other hand, so far there is a limited number of cities which offer support for these sensors. In order to deal with the lack of traffic data, a traffic data simulator component was developed in this project, so as to facilitate the experimental evaluation of the implemented system. On the other hand, there should be no doubt that embedding support for additional traffic data sources would be a great enhancement.

- **Registration of Vehicle Resources**: As it has already been mentioned, the provided bus management system is able to evaluate registered travel requests and generate timetable dynamically. Moreover, the administrator of the system has the opportunity to to make decisions regarding the number of generated timetables and the number of operating bus vehicles, which are required in order to transport the passengers of each bus line. However, it would be a notable addition to the project if the administrator was able to register the available vehicle resources in advance, so as the timetables of each bus line to be generated accordingly. In this way, transport companies which are already operating and might be interested in using the implemented system, would be able to register their already existing fleet and no adjustments would be required before generating timetables for their bus lines.

- **Registration of Travel Requests**: In this project a registered travel request document contains a unique departure datetime value, corresponding to the departure datetime preference of the passenger. Furthermore, while evaluating a set of registered travel request documents, the timetable generation algorithm utilizes the mean value of their corresponding departure datetime values, so as to identify a unique value which minimizes the average waiting time of the passengers. Nevertheless, some already existing transport systems use two kinds of travel requests, for passengers who may want to *depart after* or *arrive before* a specified datetime. Unfortunately, this distinction is not supported in the current version of the project and would be a useful addition to the timetable generation algorithm.

- **Front-end Applications**: In contrast with similar research projects, such as *Mobile Network Assisted Driving (MoNAD)*, the current project does not include any front-end applications which could be used by passengers, drivers, or system administrators and would be a remarkable upgrade.

References

[1] Stuart Lloyd. "Least Squares Quantization in PCM". *IEEE Transactions on Information Theory*, 28(2):129–137, 1982. doi:10.1109/TIT.1982.1056489.

[2] Greg Hamerly and Charles Elkan. "Alternatives to the K-Means Algorithm that Find Better Clusterings". *Proceedings of the Eleventh International Conference on Information and Knowledge Management (CIKM)*, 2002.

[3] Andrea Vattani. "K-Means Requires Exponentially Many Iterations Even in the Plane". *Discrete and Computational Geometry*, 45(4):596–616, 2011. doi:10.1007/s00454-011-9340-1.

[4] David Arthur, Bodo Manthey, and Heiko Roeglin. "k-Means has Polynomial Smoothed Complexity". *Proceedings of the 50th Symposium on Foundations of Computer Science (FOCS)*, 2009.

[5] Ahamed Shafeeq B M and Hareesha K S. "Dynamic Clustering of Data with Modified K-Means Algorithm". In *International Conference on Information and Computer Networks (ICICN 2012)*, Singapore, 2012.

[6] Edward F. Moore. "The Shortest Path through a Maze". In *Proceedings of the International Symposium on the Theory of Switching*, pages 285–292. Harvard University Press, 1959.

[7] C. Y. Lee. "An Algorithm for Path Connections and Its Applications". *IRE Transactions on Electronic Computers*, EC-10(3):346–365, Sept. 1961.

[8] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *"Introduction to Algorithms"*. MIT Press, 3rd edition, 2009.

[9] Edsger W. Dijkstra. "A Note on Two Problems in Connexion with Graphs". *Numerische Mathematik*, 1:269–271, 1959. doi:10.1007/BF01386390.

[10] Michael L. Fredman and Robert E. Tarjan. "Fibonacci Heaps And Their Uses In Improved Network Optimization Algorithms". In *25th Annual Symposium on Foundations of Computer Science*, pages 338–346. IEEE, Oct. 1984. doi:10.1109/SFCS.1984.715934.

[11] Kurt Mehlhorn and Peter Sanders. *"Algorithms and Data Structures: The Basic Toolbox"*. Springer, 2008.

[12] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, July 1968. doi: 10.1109/TSSC.1968.300136.

[13] Michael J. De Smith, Michael F. Goodchild, and Paul A. Longley. *"Geospatial Analysis: A Comprehensive Guide to Principles, Techniques and Software Tools"*. Troubadour Publishing Ltd, 2007. ISBN 9781905886609.

[14] Magnus Lie Hetland. *"Python Algorithms: Mastering Basic Algorithms in the Python Language"*. Apress, 2010. ISBN 9781430232377.

[15] Stuart Russell and Peter Norvig. *" Artificial Intelligence: A Modern Approach"*. Prentice Hall, 2nd edition, 2003. ISBN 978-0137903955.

[16] Judea Pearl. *"Heuristics: Intelligent Search Strategies for Computer Problem Solving"*. Addison-Wesley, 1984. ISBN 0-201-05594-5.

[17] Dan Puiu, Payam Barnaghi, Ralf Tönjes, Daniel Kümper, Muhammad Intizar Ali, Alessandra Mileo, Josiane Xavier Parreira, Marten Fischer, Sefki Kolozali, Nazli Farajidavar, Feng Gao, Thorben Iggena, Thu-Le Pham, Cosmin-Septimiu Nechifor, Daniel Puschmann, and Joao Fernandes. "CityPulse: Large Scale Data Analytics Framework for Smart Cities". *IEEE Access*, 4:1086–1108, 05 April 2016. doi: 10.1109/ACCESS.2016.2541999.

[18] CityPulse Consortium. "Real-Time IoT Stream Processing and Large-scale Data Analytics for Smart City Applications ". *Project Final Report*. [Online]. Available: http://www.ict-citypulse.eu/page/content/publications (Date last accessed July 27, 2017).

[19] Muhammad Intizar Ali, Feng Gao, and Alessandra Mileo. "CityBench: A Configurable Benchmark to Evaluate RSP Engines Using Smart City Datasets". In *Proceedings of ISWC 2015 - 14th International Semantic Web Conference*, pages 374–389, Bethlehem, PA, USA, 2015. W3C.

[20] *CityPulse Dataset Collection*. [Online]. Available: http://iot.ee.surrey.ac.uk:8080/ (Date last accessed July 27, 2017).

[21] *OpenStreetMap*. [Online]. Available: https://www.openstreetmap.org/ (Date last accessed July 27, 2017).

[22] *MongoDB*. [Online]. Available: https://www.mongodb.com/ (Date last accessed July 27, 2017).

[23] *BSON*. [Online]. Available: http://bsonspec.org/ (Date last accessed July 27, 2017).

[24] *JSON*. [Online]. Available: http://json.org/ (Date last accessed July 27, 2017).

[25] *Gunicorn*. [Online]. Available: http://gunicorn.org/ (Date last accessed July 27, 2017).

[26] *Python*. [Online]. Available: https://www.python.org/ (Date last accessed July 27, 2017).

[27] *HTTP - Hypertext Transfer Protocol*. [Online]. Available: https://www.w3.org/Protocols/ (Date last accessed July 27, 2017).

[28] Phillip J. Eby. "Python Web Server Gateway Interface v1.0". *Python Enhancement Proposals (PEPs): 333*, 7 Dec. 2003. [Online]. Available: https://www.python.org/dev/peps/pep-0333/ (Date last accessed July 27, 2017).

[29] *Mobile Network Assisted Driving (MoNAD)*. [Online]. Available: https://github.com/EricssonResearch/monad (Date last accessed July 27, 2017).

[30] *Android*. [Online]. Available: https://www.android.com/ (Date last accessed July 27, 2017).

[31] Pengfei Zhou, Yuanqing Zheng, and Mo Li. "How Long to Wait? Predicting Bus Arrival Time With Mobile Phone Based Participatory Sensing". *IEEE Transactions on Mobile Computing*, 13(6):1228–1241, 18 October 2013. doi: 10.1109/TMC.2013.136.

[32] *imposm.parser*. [Online]. Available: https://imposm.org/docs/imposm.parser/latest/index.html (Date last accessed July 27, 2017).

[33] Darrell Whitley. "A Genetic Algorithm Tutorial". *Statistics and Computing*, 4(2):65–85, June 1994. doi: 10.1007/BF00175354.

Appendix A

Description of Documents

A.1 MongoDB Database Documents

A.1.1 Address

address_document

```
address_document: {
    _id, name, node_id, point: {longitude, latitude}
}
```

A.1.2 Bus Line

bus_line_document

```
bus_line_document: {
    _id, line_id,
    bus_stops: [{
        _id, osm_id, name, point: {longitude, latitude}
    }]
}
```

A.1.3 Bus Stop

bus_stop_document

```
bus_stop_document: {
    _id, osm_id, name, point: {longitude, latitude}
}
```

A.1.4 Bus Stop Waypoints

bus_stop_waypoints_document

```
bus_stop_waypoints_document: {
    _id,
    starting_bus_stop: {
        _id, osm_id, name, point: {longitude, latitude}
    },
    ending_bus_stop: {
        _id, osm_id, name, point: {longitude, latitude}
    },
    waypoints: [[edge_object_id]]
}

detailed_bus_stop_waypoints_document: {
    _id,
    starting_bus_stop: {
        _id, osm_id, name, point: {longitude, latitude}
    },
    ending_bus_stop: {
        _id, osm_id, name, point: {longitude, latitude}
    },
    waypoints: [[edge_document]]
}
```

A.1.5 Bus Vehicle

bus_vehicle_document

```
bus_vehicle_document: {
    _id, bus_vehicle_id, maximum_capacity,
    routes: [{
        starting_datetime, ending_datetime, timetable_id
    }]
}
```

A.1.6 Edge

edge_document

```
edge_document: {
    _id, starting_node: {osm_id, point: {longitude, latitude}},
```

```
    ending_node: {osm_id, point: {longitude, latitude}},
    max_speed, road_type, way_id, traffic_density
}
```

A.1.7 Node

node_document

```
node_document: {
    _id, osm_id, tags, point: {longitude, latitude}
}
```

A.1.8 Point

point_document

```
point_document: {
    _id, osm_id, point: {longitude, latitude}
}
```

A.1.9 Timetable

timetable_document

```
timetable_document: {
    _id, timetable_id, line_id, bus_vehicle_id,
    timetable_entries: [{
        starting_bus_stop: {
            _id, osm_id, name, point: {longitude, latitude}
        },
        ending_bus_stop: {
            _id, osm_id, name, point: {longitude, latitude}
        },
        departure_datetime, arrival_datetime, total_time,
        number_of_onboarding_passengers,
        number_of_deboarding_passengers,
        number_of_current_passengers
    }],
    travel_requests: [{
        _id, travel_request_id, client_id, bus_line_id,
        starting_bus_stop: {
            _id, osm_id, name, point: {longitude, latitude}
```

```
        },
        ending_bus_stop: {
            _id, osm_id, name, point: {longitude, latitude}
        },
        departure_datetime, arrival_datetime,
        starting_timetable_entry_index,
        ending_timetable_entry_index
    }]
}
```

A.1.10 Traffic Event

traffic_event_document

```
traffic_event_document: {
    _id, event_id, event_type, event_level,
    point: {longitude, latitude}, datetime
}
```

A.1.11 Travel Request

travel_request_document

```
travel_request_document: {
    _id, client_id, line_id,
    starting_bus_stop: {
        _id, osm_id, name, point: {longitude, latitude}
    },
    ending_bus_stop: {
        _id, osm_id, name, point: {longitude, latitude}
    },
    departure_datetime, arrival_datetime,
    starting_timetable_entry_index,
    ending_timetable_entry_index
}
```

A.1.12 Way

way_document

```
way_document: {
    _id, osm_id, tags, references
}
```

A.2 Route Generator Responses

A.2.1 Less Time-Consuming Route Between Two Bus Stops

get_route_between_two_bus_stops

```
get_route_between_two_bus_stops_response: {
    starting_bus_stop: {
        _id, osm_id, name, point: {longitude, latitude}
    },
    ending_bus_stop: {
        _id, osm_id, name, point: {longitude, latitude}
    },
    route: {
        total_distance, total_time, node_osm_ids, points, edges,
        distances_from_starting_node, times_from_starting_node,
        distances_from_previous_node, times_from_previous_node
    }
}
```

A.2.2 Less Time-Consuming Route Between Multiple Bus Stops

get_route_between_multiple_bus_stops

```
get_route_between_multiple_bus_stops_response: [{
    starting_bus_stop: {
        _id, osm_id, name, point: {longitude, latitude}
    },
    ending_bus_stop: {
        _id, osm_id, name, point: {longitude, latitude}
    },
    route: {
        total_distance, total_time, node_osm_ids, points, edges,
        distances_from_starting_node, times_from_starting_node,
        distances_from_previous_node, times_from_previous_node
```

```
    }
}]
```

A.2.3 All Possible Routes Between Two Bus Stops

get_waypoints_between_two_bus_stops

```
get_waypoints_between_two_bus_stops_response: {
    starting_bus_stop: {
        _id, osm_id, name, point: {longitude, latitude}
    },
    ending_bus_stop: {
        _id, osm_id, name, point: {longitude, latitude}
    },
    waypoints: [[{
        _id, starting_node: {osm_id, point: {longitude, latitude}},
        ending_node: {osm_id, point: {longitude, latitude}},
        max_speed, road_type, way_id, traffic_density
    }]]
}
```

A.2.4 All Possible Routes Between Multiple Bus Stops

get_waypoints_between_multiple_bus_stops

```
get_waypoints_between_multiple_bus_stops_response: [{
    starting_bus_stop: {
        _id, osm_id, name, point: {longitude, latitude}
    },
    ending_bus_stop: {
        _id, osm_id, name, point: {longitude, latitude}
    },
    waypoints: [[{
        _id, starting_node: {osm_id, point: {longitude, latitude}},
        ending_node: {osm_id, point: {longitude, latitude}},
        max_speed, road_type, way_id, traffic_density
    }]]
}]
```

A.3 CityPulse Data Bus Outputs

A.3.1 Traffic Jam Event

traffic_jam_event

```
[{
    subject: ‘http://purl.oclc.org/NET/UNIS/sao/sao#cc5fe3f9-2a52-4134-a85b’,
    predicate: ‘http://www.w3.org/1999/02/22-rdf-syntax-ns#type’,
    object: ‘http://purl.oclc.org/NET/UNIS/sao/ec#TrafficJam’,
    graph: ‘’
 },
 {
    subject: ‘http://purl.oclc.org/NET/UNIS/sao/sao#cc5fe3f9-2a52-4134-a85b’,
    predicate: ‘http://purl.oclc.org/NET/UNIS/sao/ec#hasSource’,
    object: ‘"USER_testuser"’,
    graph: ‘’
 },
 {
    subject: ‘http://purl.oclc.org/NET/UNIS/sao/sao#cc5fe3f9-2a52-4134-a85b’,
    predicate: ‘http://purl.oclc.org/NET/UNIS/sao/sao#hasLevel’,
    object: ‘"2"^^http://www.w3.org/2001/XMLSchema#long’,
    graph: ‘’
 },
 {
    subject: ‘_:b0’,
    predicate: ‘http://www.w3.org/1999/02/22-rdf-syntax-ns#type’,
    object: ‘http://www.w3.org/2003/01/geo/wgs84_pos#Instant’,
    graph: ‘’
 },
 {
    subject: ‘_:b0’,
    predicate: ‘http://www.w3.org/2003/01/geo/wgs84_pos#lat’,
    object: ‘"56.1155615424"^^http://www.w3.org/2001/XMLSchema#double’,
    graph: ‘’
 },
 {
    subject: ‘_:b0’,
    predicate: ‘http://www.w3.org/2003/01/geo/wgs84_pos#lon’,
    object: ‘"10.1870405674"^^http://www.w3.org/2001/XMLSchema#double’,
    graph: ‘’
 },
 {
    subject: ‘http://purl.oclc.org/NET/UNIS/sao/sao#cc5fe3f9-2a52-4134-a85b’,
    predicate: ‘http://purl.oclc.org/NET/UNIS/sao/sao#hasLocation’,
    object: ‘_:b0’,
    graph: ‘’
 },
 {
    subject: ‘http://purl.oclc.org/NET/UNIS/sao/sao#cc5fe3f9-2a52-4134-a85b’,
    predicate: ‘http://purl.oclc.org/NET/UNIS/sao/sao#hasType’,
    object: ‘http://purl.oclc.org/NET/UNIS/sao/ec#TransportationEvent’,
    graph: ‘’
 },
 {
    subject: ‘http://purl.oclc.org/NET/UNIS/sao/sao#cc5fe3f9-2a52-4134-a85b’,
    predicate: ‘http://purl.org/NET/c4dm/timeline.owl#time’,
```

```
    object: '"2016-10-11T11:13:21.000Z"^^http://www.w3.org/2001/XMLSchema#dateTime',
    graph: ''
  }
]
```

Appendix B

Pseudocodes

B.1 Route Identification and Evaluation Algorithm

Python Pseudocode

```
1: function identify_path_with_lowest_cost(start, end, edges_dictionary):
       """
       This function is capable of identifying the path with
       the lowest cost value (less time-consuming in this case)
       connecting the starting with the ending node, implementing
       a variation of the A* search algorithm.

       edge_document: {
           _id, starting_node: {osm_id, point: {longitude, latitude}},
           ending_node: {osm_id, point: {longitude, latitude}},
           max_speed, road_type, way_id, traffic_density
       }
       :param start: {osm_id, point: {longitude, latitude}}
       :param end: {osm_id, point: {longitude, latitude}}
       :param edges_dictionary: {starting_node_osm_id -> [edge_document]}

       :return: path_between_two_nodes: {
                    total_distance, total_time, node_osm_ids, points, edges,
                    distances_from_starting_node, times_from_starting_node,
                    distances_from_previous_node, times_from_previous_node
                }

               (None if there is no path between the provided nodes)
       """
       # A dictionary ({node_osm_id -> node}) containing nodes that have
       # already been evaluated (cost values have been estimated).
2:     closed_set = {}

       # A node storing structure containing nodes whose neighbors
       # should be evaluated (cost values should be estimated).
3:     open_set = OrderedSet()

       # Initialize starting_node.
4:     starting_node_osm_id = start.get(osm_id)
5:     starting_node_point_document = start.get(point)
6:     starting_node = Node(
           osm_id=starting_node_osm_id,
```

```
            point_document=starting_node_point_document
        )

        # Initialize ending_node.
 7:     ending_node_osm_id = end.get(osm_id)
 8:     ending_node_point_document = end.get(point)
 9:     ending_node = Node(
            osm_id=ending_node_osm_id,
            point_document=ending_node_point_document
        )

        # Real cost values of starting_node are equal to zero.
10:     starting_node.set_real_cost(
            real_distance_cost=0.0,
            real_travelling_time_cost=0.0
        )

        # Estimate heuristic cost values of starting_node.
11:     heuristic_distance_cost, heuristic_travelling_time_cost =
            estimate_heuristic_cost(
                starting_point_document=starting_node.point_document,
                ending_point_document=ending_node.point_document
        )
12:     starting_node.set_heuristic_cost(
            heuristic_distance_cost=heuristic_distance_cost,
            heuristic_travelling_time_cost=heuristic_travelling_time_cost
        )

        # Estimate total score values of starting_node.
13:     starting_node.estimate_total_score()

        # Set the followed_path of starting_node.
14:     starting_node.add_node_to_followed_path(node=starting_node)

        # Add the starting_node to closed_set, since it has already been evaluated.
15:     closed_set[starting_node_osm_id] = starting_node

        # Add the starting_node to open_set, since its neighbors should be evaluated.
16:     open_set.insert(new_node=starting_node)

        # The node with the lowest total_score value is retrieved from the open_set,
        # as long as the number of stored nodes is greater than zero.
17:     while len(open_set) > 0:

            # At the first iteration of this loop, starting_node is retrieved.
18:         current_node = open_set.pop()

            # Ending condition: ending_node has been discovered =>
            # followed path should be processed in order to retrieve
            # its parameters (covered distance, travelling time,
            # intermediate nodes, points, and edges).
19:         if current_node.osm_id == ending_node_osm_id:
20:             return process_followed_path(
                    list_of_nodes=current_node.get_followed_path(),
                    edges_dictionary=edges_dictionary
                )
```

```
            # Continuation condition: current_node has no neighbors.
21:         if current_node.osm_id not in edges_dictionary:
22:             continue

            # Case that current_node has neighbors. Each neighbor should be evaluated,
            # taking into consideration the parameters of corresponding edges.
23:         for edge in edges_dictionary.get(current_node.osm_id):
24:             next_node_osm_id = edge.get(ending_node).get(osm_id)
25:             next_node_point_document = edge.get(ending_node).get(point)
26:             max_speed = edge.get(max_speed)
27:             road_type = edge.get(road_type)
28:             traffic_density = edge.get(traffic_density)

                # Check whether next_node has already been evaluated.
29:             if next_node_osm_id in closed_set:
30:                 next_node = closed_set.get(next_node_osm_id)
31:             else:
                    # Case that next_node has not been evaluated
32:                 next_node = Node(
                        osm_id=next_node_osm_id,
                        point_document=next_node_point_document
                    )

                    # Heuristic cost should be estimated.
33:                 heuristic_distance_cost, heuristic_travelling_time_cost =
                        estimate_heuristic_cost(
                            starting_point_document=next_node.point_document,
                            ending_point_document=ending_node.point_document
                        )
34:                 next_node.set_heuristic_cost(
                        heuristic_distance_cost=heuristic_distance_cost,
                        heuristic_travelling_time_cost=heuristic_travelling_time_cost
                    )

                # Estimate the real cost values for travelling
                # from current_node to next_node.
35:             additional_real_distance_cost, additional_real_travelling_time_cost =
                    estimate_real_cost(
                        starting_point_document=current_node.point_document,
                        ending_point_document=next_node.point_document,
                        max_speed=max_speed,
                        road_type=road_type,
                        traffic_density=traffic_density
                    )

                # Estimate the real cost values for travelling
                # from starting_node to next_node.
36:             new_real_distance_cost = current_node.real_distance_cost
                                       + additional_real_distance_cost
37:             new_real_travelling_time_cost = current_node.real_travelling_time_cost
                                              + additional_real_travelling_time_cost

                # Continuation condition: Compare newly estimated real cost values
                # with previous ones (in case next_node was evaluated again).
                # If the previously followed_path has lower cost value,
                # then the loop continues evaluating the next neighbor.
                # (Non-evaluated nodes are initialized with real cost
```

```
                # values equal to infinity).
38:             if next_node.real_travelling_time_cost < new_real_travelling_time_cost:
39:                 continue

                # Real cost values of next_node should be replaced.
40:             next_node.set_real_cost(
                    real_distance_cost=new_real_distance_cost,
                    real_travelling_time_cost=new_real_travelling_time_cost
                )

                # Total cost values of next_nodes are estimated.
41:             next_node.estimate_total_score()

                # Followed path of next_node is updated.
42:             next_node.set_followed_path(
                    followed_path=current_node.get_followed_path() + [next_node]
                )

                # Since it has been evaluated, next_node is added to the closed_set.
43:             closed_set[next_node_osm_id] = next_node

                # Add next_node to the open_set, so as to allow
                # its neighbors to be evaluated.
44:             if not open_set.exists(next_node_osm_id):
45:                 open_set.insert(new_node=next_node)

        # There is no path connecting start with end.
46:     return None
47: end function
```

B.2 Waypoints Identification Algorithm

Python Pseudocode

```
 1: function identify_all_paths(starting_node_osm_id,
                                ending_node_osm_id,
                                edges_dictionary):
        """
        This function is capable of identifying all the possible paths
        connecting the starting with the ending node, implementing a
        variation of the Breadth-first search algorithm. Each path is
        represented by a list of edge_documents (waypoints), including
        details about intermediate nodes, maximum allowed speed, road
        type, and current levels of traffic density. The returned value
        of the function is a double list of edge_documents.

        edge_document: {
            _id, starting_node: {osm_id, point: {longitude, latitude}},
            ending_node: {osm_id, point: {longitude, latitude}},
            max_speed, road_type, way_id, traffic_density
        }
        :param starting_node_osm_id: integer
        :param ending_node_osm_id: integer
        :param edges_dictionary: {starting_node_osm_id -> [edge_document]}
        :return: waypoints: [[edge_document]]
        """
```

```
        # Returned value
 2:     waypoints = []

        # A data storing structure used in order to keep the nodes
        # whose neighbors should be considered.
 3:     open_set = MultiplePathsSet()

        # A dictionary ({node_osm_id -> node}) containing nodes
        # whose neighbors have already been considered.
 4:     closed_set = {}

        # starting_node is initialized and pushed into the open_set.
 5:     starting_node = MultiplePathsNode(osm_id=starting_node_osm_id)
 6:     starting_node.followed_paths = [[starting_node.osm_id]]
 7:     open_set.push(new_node=starting_node)

        # The node in the first position of the open_set is retrieved,
        # as long as the number of stored nodes is above zero.
 8:     while len(open_set) > 0:
 9:         current_node = open_set.pop()

            # Continuation condition: ending_node has been discovered.
10:         if current_node.osm_id == ending_node_osm_id:

                # Each one of the followed paths is processed, in order
                # to retrieve the corresponding edge_documents,
                # and added to the returned double list.
11:             for followed_path in current_node.get_followed_paths():
12:                 waypoints.append(process_followed_path(
                        followed_path=followed_path,
                        edges_dictionary=edges_dictionary)
                    )

13:             current_node.followed_paths = []
14:             continue

            # Continuation condition: current_node is ignored in case
            # it has no neighbors, or its neighbors have already
            # been considered.
15:         if (current_node.osm_id not in edges_dictionary or
                current_node.osm_id in closed_set):
16:             continue

            # Following the edges of current_node, each one
            # of its neighbors is considered.
17:         for edge in edges_dictionary.get(current_node.osm_id):
18:             next_node_osm_id = edge.get(ending_node).get(osm_id)

                # Continuation condition: next_node has already been considered.
19:             if next_node_osm_id in closed_set:
20:                 continue
21:             else:
                    # Followed paths of next_node are updated and the node
                    # is pushed into the open_set, so as to allow its
                    # neighbors to be considered.
22:                 next_node = MultiplePathsNode(osm_id=next_node_osm_id)
23:                 next_node.update_followed_paths(
```

```
                        followed_paths_of_previous_node=
                            current_node.get_followed_paths()
                    )
24:                 open_set.push(new_node=next_node)

            # Since all its neighbors have been considered,
            # current_node is added to the closed_set.
25:         closed_set[current_node.osm_id] = current_node

26:     return waypoints
27: end function
```

B.3 Timetable Generation Algorithm

Python Pseudocode

```
1: function generate_timetables_for_bus_line(bus_line, timetables_starting_datetime,
                                              timetables_ending_datetime,
                                              travel_requests_min_departure_datetime,
                                              travel_requests_max_departure_datetime):
       """
       This function is capable of generating timetables for a bus_line,
       while evaluating travel_requests of a specific datetime period.

       - The input: timetables_starting_datetime and input: timetables_ending_
         datetime are provided to the function, so as timetables for the
         specific datetime period to be generated.

       - The input: requests_min_departure_datetime and input: requests_max_
         departure_datetime are provided to the function, so as travel_
         requests with departure_datetime corresponding to the
         specific datetime period to be evaluated.

       - The input: bus_line or input: line_id is provided to the function,
         so as timetables for the specific bus_line to be generated.

       bus_stop_document: {
           _id, osm_id, name, point: {longitude, latitude}
       }
       bus_line_document: {
           _id, line_id, bus_stops: [bus_stop_document]
       }
       travel_request_document: {
           _id, client_id, line_id,
           starting_bus_stop: bus_stop_document,
           ending_bus_stop: bus_stop_document,
           departure_datetime, arrival_datetime,
           starting_timetable_entry_index, ending_timetable_entry_index
       }
       timetable_document: {
           _id, line_id,
           timetable_entries: [{
               starting_bus_stop: bus_stop_document,
               ending_bus_stop: bus_stop_document,
               departure_datetime, arrival_datetime,
               number_of_onboarding_passengers,
```

```
            number_of_deboarding_passengers,
            number_of_current_passengers,
            route: {
                total_distance, total_time, node_osm_ids, points, edges,
                distances_from_starting_node, times_from_starting_node,
                distances_from_previous_node, times_from_previous_node
            }
        }],
        travel_requests: [travel_request_document]
    }
    :param timetables_starting_datetime: datetime
    :param timetables_ending_datetime: datetime
    :param requests_min_departure_datetime: datetime
    :param requests_max_departure_datetime: datetime
    :param bus_line: bus_line_document
    :param line_id: int
    :return: output = {
                current_timetables: [timetable_document],
                average_waiting_time_of_current_timetables: float (in seconds),
                number_of_bus_vehicles: int
             }
    """
    # The list of bus_stops corresponding to the provided bus_line is retrieved.
2:  bus_stops = bus_line.get(bus_stops)

    # All the registered travel_requests, with departure_datetime corresponding
    # to the provided datetime period, are retrieved in order to be evaluated.
3:  travel_requests = get_travel_requests(
        bus_line,
        travel_requests_min_departure_datetime,
        travel_requests_max_departure_datetime
    )

    # The less time-consuming route connecting the specified bus_stops
    # is identified, taking into consideration current levels of
    # traffic density.
4:  route_generator_response = get_less_time_consuming_route(bus_stops)

    # Based on the parameters of identified route and supposing that only one
    # bus vehicle is used, some initial timetables are generated covering the
    # whole datetime period from timetables_starting_datetime to timetables_
    # ending_datetime. Initially, the list of travel_requests of these
    # timetables is empty, and the departure_datetime and arrival_datetime
    # values of their timetable_entries are based exclusively on the
    # parameters of te identified bus_route. In the next steps of the
    # algorithm, these timetables are used in the initial clustering
    # of the travel_requests.
5:  current_timetables = generate_initial_timetables(route_generator_response)

    # Since no travel_requests are served by initial timetables,
    # average_waiting_time is initialized to infinity.
6:  average_waiting_time_of_current_timetables = Infinity

    # Loop: new_timetables are generated, replacing current_timetables,
    # as long as the average_waiting_time of passengers is reduced.
7:  while True:
8:      new_timetables = generate_new_timetables_based_on_travel_requests(
```

```
                current_timetables,
                travel_requests
            )
 9:         average_waiting_time_of_new_timetables =
                calculate_average_waiting_time(
                    new_timetables
                )

10:         if (average_waiting_time_of_new_timetables <
                    average_waiting_time_of_current_timetables):
11:             current_timetables = new_timetables
12:             average_waiting_time_of_current_timetables =
                    average_waiting_time_of_new_timetables
13:         else:
14:             break

        # The number_of_bus_vehicles is calculated, required in order
        # to transport the passengers of produced timetables.
15:     number_of_bus_vehicles = calculate_number_of_bus_vehicles(current_timetables)

        # The output of the function includes the produced timetables, calculated
        # average_waiting_time of passengers, and number of required bus_vehicles.
16:     output = {
            current_timetables,
            average_waiting_time_of_current_timetables,
            number_of_bus_vehicles
        }
17:     return output
18: end function


19: function generate_new_timetables_based_on_travel_requests(current_timetables,
                                                              travel_requests):
        """
        This function is capable of generating new_timetables, evaluating
        a list of travel_requests. Initially, the timetables_entries of
        new_timetables are based on the corresponding timetable_entries
        of current_timetables, and  their travel_requests entry is empty.
        In the next processing steps, each travel_request is corresponded
        to the new_timetable which produces the minimum_individual_waiting_time
        for the passenger, while updating in parallel the timetable_entries
        of new_timetables.

        travel_request_document: {
            _id, client_id, line_id,
            starting_bus_stop: bus_stop_document,
            ending_bus_stop: bus_stop_document,
            departure_datetime, arrival_datetime,
            starting_timetable_entry_index, ending_timetable_entry_index
        }
        timetable_document: {
            _id, line_id,
            timetable_entries: [{
                starting_bus_stop: bus_stop_document,
                ending_bus_stop: bus_stop_document,
                departure_datetime, arrival_datetime,
                number_of_onboarding_passengers,
```

```
                number_of_deboarding_passengers,
                number_of_current_passengers,
                route: {
                    total_distance, total_time, node_osm_ids, points, edges,
                    distances_from_starting_node, times_from_starting_node,
                    distances_from_previous_node, times_from_previous_node
                }
            }],
            travel_requests: [travel_request_document]
        }
        :param current_timetables: [timetable_document]
        :param travel_requests: [travel_request_document]
        :return: new_timetables: [timetable_document]
        """
        # The timetables_entries of new_timetables are initialized, based on
        # the corresponding timetable_entries of current_timetables, and
        # their travel_requests entry is empty.
20:     new_timetables = generate_empty_timetables()
21:     new_timetables.set(timetable_entries) =
            current_timetables.get(timetable_entries)

        # (Initial Clustering): Each one of the provided travel_requests is
        # corresponded to the timetable which ensures the minimum_individual_
        # waiting_time for the passenger. Waiting time is calculated as the
        # difference between the departure_datetime of the travel_request
        # and the departure_datetime of the timetable_entry from where the
        # passenger departs from (identified by the starting_timetable_
        # entry_index value).
22:     correspond_travel_requests_to_timetables(travel_requests, new_timetables)

        # The number of onboarding, deboarding, and current passengers are
        # calculated, for each timetable entry in each one of the new_timatables.
23:     calculate_number_of_passengers_of_timetables(new_timetables)

        # For each one of the new_timetables, the departure_datetime and arrival_
        # datetime values of each timetable_entry are re-estimated, taking into
        # consideration the departure_datetime values of the corresponding
        # travel_requests. In each timetable and for each travel_request, the
        # ideal departure_datetimes from all bus_stops (not only the departing
        # bus_stop of passenger) are estimated. Then, the ideal departure_datetimes
        # of the timetable, for each bus stop, correspond to the mean values of the
        # ideal departure_datetimes of the corresponding travel_requests. Finally,
        # starting from the initial bus_stop and combining the ideal departure_
        # datetimes of each bus_stop and the required traveling time between
        # bus_stops, included in the parameters of the identified bus_route,
        # the departure_datetimes of the timetable_entries are finalized.
24:     adjust_departure_and_arrival_datetimes_of_timetables(new_timetables)

        # (Handling of Undercrowded Timetables): After the initial clustering step,
        # there might be timetables where the number of corresponded travel_requests
        # is lower than the minimum_number_of_passengers_in_timetable value
        # (threshold provided by administrator). This is usual during night hours,
        # where transportation demand is not so high. These timetables are removed
        # from the list of new_timetables and each one of their travel_requests
        # is corresponded to one of the remaining timetables, based on the
        # individual_waiting_time of the passenger (minimum).
25:     handle_undercrowded_timetables(new_timetables)
```

```
26:     calculate_number_of_passengers_of_timetables(new_timetables)
27:     adjust_departure_and_arrival_datetimes_of_timetables(new_timetables)

        # (Handling of Overcrowded Timetables): In addition, there might be timetables
        # where the number_of_current_passengers is higher than the maximum_bus_
        # capacity (threshold provided by the administrator), indicating that
        # the timetable cannot be served by one bus vehicle. In this case, each
        # one of these timetables should be divided into two timetables, and
        # the corresponding travel_requests should be partitioned. The whole
        # procedure is repeated until there is no timetable where the
        # number_of_current_passengers exceeds the maximum_bus_capacity.
28:     handle_overcrowded_timetables(new_timetables)
29:     calculate_number_of_passengers_of_timetables(new_timetables)
30:     adjust_departure_and_arrival_datetimes_of_timetables(new_timetables)

        # (Average Waiting Time): For each new_timetable, the average_waiting_time
        # of passengers is calculated. If it is higher than the average_waiting_
        # time_threshold (provided by administrator), then the possibility of
        # dividing the timetable is investigated. If the two new timetables
        # demonstrate lower average_waiting_time than the initial one and both
        # serve more travel_requests than the minimum_number_of_passengers_
        # in_timetable (threshold provided by the administrator), then the
        # initial timetable is divided, its travel_requests are partitioned,
        # and the departure_datetime and arrival_datetime values in the
        # timetable_entries of new timetables, are re-estimated based on
        # the departure_datetime values of the partitioned travel_requests.
31:     handle_timetables_with_average_waiting_time_above_threshold(new_timetables)
32:     calculate_number_of_passengers_of_timetables(new_timetables)
33:     adjust_departure_and_arrival_datetimes_of_timetables(new_timetables)

34:     return new_timetables
35: end function
```

Appendix C

Technical Instructions

C.1 System Requirements and Deployment Instructions

Operating System: Linux (Tested on Ubuntu 16.04)

Programming Languages:

- **Python:** Tested on Python 2.7.12
- **JavaScript:** Tested on Node.js 6.7.0

Tools – Packages – Libraries:

1. **pip:** The PyPA recommended tool for installing Python packages.

   Installation: https://pip.pypa.io/en/stable/installing/

   For Ubuntu 16.04: `$ sudo apt-get install python-pip`

2. **NumPy:** The fundamental package for scientific computing with Python.

   Installation: http://www.scipy.org/scipylib/download.html

   For Ubuntu 16.04: `$ sudo pip install numpy`

3. **Requests:** An elegant and simple HTTP library for Python.

   Installation: http://docs.python-requests.org/en/master/user/install/#install

   For Ubuntu 16.04: `$ sudo pip install requests`

4. **Imposm:** An importer for OpenStreetMap data which reads XML and PBF files.

   Installation: https://imposm.org/docs/imposm/latest/install.html

   For Ubuntu 16.04: `$ sudo pip install imposm`

5. **Gunicorn:** "Green Unicorn" is a Python WSGI HTTP Server for UNIX, broadly compatible with various web frameworks, simply implemented, light on server resources, and fairly speedy.

   Installation: http://gunicorn.org/

   For Ubuntu 16.04: `$ sudo pip install gunicorn`

6. **Greenlet:** A Python package which introduces a primitive notion of micro-threads, with no implicit scheduling; coroutines, in other words.

   Installation: https://greenlet.readthedocs.io/en/latest/

   For Ubuntu 16.04: `$ sudo pip install greenlet`

7. **Eventlet:** A concurrent networking library for Python.

   Installation: http://eventlet.net/

   For Ubuntu 16.04: `$ sudo pip install eventlet`

8. **gevent:** A coroutine-based Python networking library that uses greenlet to provide a high-level synchronous API on top of the libev event loop.

   Installation: http://www.gevent.org/

   For Ubuntu 16.04: `$ sudo pip install gevent`

9. **Node.js:** A JavaScript runtime built on Chrome's V8 JavaScript engine, which uses an event-driven, non-blocking I/O model that makes it lightweight and efficient.

   Installation: https://nodejs.org/

   For Ubuntu 16.04: `$ sudo apt-get install nodejs`

10. **npm:** Node.js' package ecosystem, npm, is the largest ecosystem of open source libraries in the world.

    For Ubuntu 16.04: `$ sudo apt-get install npm`

11. **N3.js:** Lightning fast, asynchronous, streaming Turtle / N3 / RDF library.

    Installation: https://www.npmjs.com/package/n3

    For Ubuntu 16.04: `$ sudo npm install n3`

12. **amqplib:** An AMQP 0-9-1 (e.g., RabbitMQ) library and client.

    Installation: https://www.npmjs.com/package/amqplib

    For Ubuntu 16.04: `$ sudo npm install amqplib`

13. **MongoDB:** An open-source, document database designed for ease of development and scaling. Tested on MongoDB shell version 3.2.10.

    Installation: https://docs.mongodb.com/manual/installation/

14. **MongoDB/Python Connector (pymongo):** The official MongoDB driver for Python.

    Installation: http://api.mongodb.com/python/current/installation.html

    For Ubuntu 16.04: `$ sudo pip install pymongo`

15. **MongoDB/Node.js Connector (mongodb):** The official MongoDB driver for Node.js.

    Installation: https://www.npmjs.com/package/mongodb

    For Ubuntu 16.04: `$ sudo npm install mongodb`

C.2 Testing Instructions

Directories:

- src: Contains the coding files of the System Database, Look Ahead, Route Generator, OSM Parser, Data Simulator (Travel Requests Simulator and Traffic Data Simulator), and Traffic Data Parser components.
- resources/osm_files: Used in order to store the input OpenStreetMap files.
- CityPulseIntegration: Contains coding files for connecting to the CityPulse Data Bus, retrieving traffic data events, and storing them to the System Database.
- tests: Contains testing files for whole the application or individual components.

MongoDB – Running: Before initializing the components of the system and running the testing files, MongoDB should be running. Detailed instructions of how to start or stop MongoDB could be found following the official documentation link:

- *https://docs.mongodb.com/v3.2/tutorial/install-mongodb-on-ubuntu/*

System Parameters – Initialization: Before running the testing files, some system parameters, such as the host and port of the System Database and Route Generator for example, should be initialized. These parameters are available at the following file:

- src/common/parameters.py

Route Generator – Initialization: Before running the Route Generator, some of its parameters, such as the number of gunicorn workers and the number of requests which are kept in the backlog when every worker is busy, should be initialized. These parameters are available at the following file:

- src/route_generator/route_generator_server_settings.py

Route Generator – Running: Before running the testing files, the Route Generator should be running. This could be done running the following command, which starts **gunicorn** using the following files:

- **Main**: src/route_generator/route_generator_server.py
- **Configuration**: src/route_generator/route_generator_server_settings.py

```
dynamic-bus-scheduling/src/route_generator$
    gunicorn -c route_generator_server_settings.py route_generator_server
```

Testing: The tests directory includes the following testing files:

- **Application**: application_test.py
- **OSM Parser**: osm_parser_ test.py
- **Travel Requests Simulator**: travel_requests_ simulator_test.py
- **Traffic Data Simulator**: traffic_data_simulator_test.py
- **Traffic Data Parser**: traffic_data_parser_test.py
- **System Database**: mongodb_database_connection_test.py
- **Route Generator**: route_generator_test.py
- **Look Ahead**: look_ahead_ test.py

In order to run the **application_test.py** file, the following command should be used:

```
dynamic-bus-scheduling/tests$ python application_test.py
```

The **application_test** offers the following options:

1. **(mongodb_database) clear_collections**: All documents stored at the collections of System Database are deleted.
2. **(osm_parser) test_parse_osm_file**: The OSM Parser is tested by retrieving geospatial data from the provided OpenStreetMap file, corresponding to addresses, bus stops, edges, nodes, points, and ways.
3. **(osm_parser) test_populate_all_collections**: The OSM Parser is tested by storing the retrieved geospatial data to the corresponding collections of System Database.
4. **(mongodb_database) print_collections**: User has the option to print some documents of the aforementioned collections of System Database. The number of printed documents is selected by the user.
5. **(route_generator) test_get_route_between_two_bus_stops**: The Route Generator is tested by retrieving the less time-consuming route between two bus stops. The names of testing bus stops are initialized through the **testing_bus_stop_names** variable included in the src/common/parameters.py file. The provided names are corresponded to stored bus stops in the System Database.
6. **(route_generator) test_get_route_between_multiple_bus_stops**: The Route Generator is tested by retrieving the less time-consuming route between multiple bus stops.
7. **(route_generator) test_get_waypoints_between_two_bus_stops**: The Route Generator is tested by retrieving all the possible waypoints between two bus stops.
8. **(route_generator) test_get_waypoints_between_multiple_bus_stops**: The Route Generator is tested by retrieving all the possible waypoints between multiple bus stops.

9. **(look_ahead_handler) test_generate_bus_line**: The Look Ahead is tested by generating a new bus line document and store it at the corresponding collection of System Database. Moreover, the intermediate waypoints connecting the bus stops of bus line are identified, by the Route Generator, and stored as well at the System Database.

10. **(mongodb_database) print_bus_line_documents**: User has the option to print the generated bus line document.

11. **(mongodb_database) print_detailed_bus_stop_waypoints_documents**: User has the option to print the stored bus stop waypoints documents.

12. **(travel_requests_simulator) test_generate_travel_request_documents**: The Travel Requests Simulator is tested by generating a number of travel requests for a 24-hour period and store them to the corresponding collection of System Database. The total number, id of bus line, and initial datetime value of the generated travel requests are selected by the user.

13. **(mongodb_database) print_travel_request_documents**: User has the option to print some of the generated travel request documents.

14. **(look_ahead_handler) test_generate_timetables_for_bus_line**: The Look Ahead is tested by generating timetables for a bus line and store them at the corresponding collection of System Database.

15. **(mongodb_database) print_timetable_documents**: User has the option to print the generated timetable documents.

16. **(traffic_data_simulator) test_generate_traffic_data_between_multiple_stops**: Traffic Data Simulator is tested by updating the traffic density values of stored edge documents, which connect the testing bus stops.

17. **(mongodb_database) print_traffic_density_documents**: User has the option to print the edge documents with updated traffic density values.

18. **(look_ahead_handler) test_update_timetables_of_bus_line**: The Look Ahead is tested by updating the timetables corresponding to the testing bus line. A request is sent to the Route Generate in order to identify the less time-consuming route connecting the intermediate bus stops, while taking into consideration the current values of traffic density. The timetable entries are updated, based on the parameters of identified routes.

19. **(look_ahead_handler) start_timetables_generator_process***: A new process is spawned, responsible for generating new timetables periodically.

20. **(look_ahead_handler) terminate_timetables_generator_process**: The timetables generator process is terminated.

21. **(look_ahead_handler) start_timetables_updater_process***: A new process is spawned, responsible for updating the existing timetables periodically.

22. **(look_ahead_handler) terminate_timetables_updater_process**: The timetables updater process is terminated.

23. **(travel_requests_simulator) start_travel_requests_generator_process***: A new process is spawned, responsible for generating new travel requests periodically.

24. **(travel_requests_simulator) terminate_travel_requests_generator_process**: The travel requests generator process is terminated.

25. **(traffic_data_simulator) start_traffic_data_generator_process***: A new process is spawned, responsible for updating the traffic density values of edge documents periodically.

26. **(traffic_data_simulator) terminate_traffic_data_generator_process**: The traffic data generator process is terminated.

* The frequency and total duration of operations in options **19**, **21**, **23**, and **25** are provided as inputs by the user, through the src/common/parameters.py file.