Minimal Effective Time Two-Way Park and Ride Problem

BY

Sergio Spinatelli
B.S., Politecnico di Milano, Milan, Italy, February 2013

THESIS

Submitted as partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Chicago, 2015

Chicago, Illinois

Defense Committee:

Jakob Eriksson, Chair and Advisor
Ouri Wolfson
Stefano Zanero, Politecnico di Milano
ACKNOWLEDGEMENTS

This work has been made possible thanks to the many people who gave me support, guidance, inspiration and help.

First of all, I would like to thank my advisors, Prof. Edoardo Amaldi and Dr. Maurizio Bruglieri, for their guidance and encouragements. I would also like to thank my UIC advisor, Prof. Jakob Eriksson, who has been a great help in enhancing this work and relating theory to practice.

A special thank you is for my family, who supported me and encouraged me in every situation, and to Veronica Silvestri, who had the patience to put up with me all this time. A last, big thank you goes to my friends, who shared so many great experiences with me in these years and made this part of my life one to remember.
TABLE OF CONTENTS

CHAPTER                  PAGE
1  PRELIMINARIES          1
    1.1  Graph Theory       1
    1.2  Network Modeling   3
    1.2.1 Uni-modal networks 3
    1.2.2 Multi-modal networks 5

2  LITERATURE REVIEW      8
    2.1  The Shortest Path Problem 8
    2.2  Time-Independent Routing 10
    2.2.1 Uni-modal routing 11
    2.2.2 Multi-modal routing 13
    2.3  Speed-up Techniques 16
    2.4  Time-Dependent Routing 19
    2.4.1 Uni-modal routing 30
    2.4.2 Multi-modal routing 32
    2.4.3 Access-Node Routing 33
    2.4.4 Core-based Access-Node Routing 34
    2.5  Two Way Shortest Path Search 36
    2.5.1 Enumerative approach 37
    2.5.2 Regular language constrained approach 38

3  PROBLEM DEFINITION     40
    3.1  Effective Time Two-Way Park-aNd-Ride Problem 40
    3.2  Main features 44

4  PROPOSED APPROACH      52
    4.1  Proposed Approach 52
    4.1.1 Details on Access-nodes 54
    4.1.2 Extension to Park-aNd-Ride: PNR-Nodes 57
    4.2  Preprocessing 59
    4.2.1 Known methods for computing access-nodes 60
    4.2.2 Determining PNR-Nodes 63
    4.3  Search for an optimal two-way path 66

5  IMPLEMENTATION AND EXPERIMENTAL RESULTS 83
    5.1  OpenTripPlanner 83
    5.2  Implementation 84
# TABLE OF CONTENTS (Continued)

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.2.1</td>
<td>Graph preprocessing</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Two-way search algorithm implementation</td>
</tr>
<tr>
<td>5.3</td>
<td>Input Data</td>
</tr>
<tr>
<td>5.4</td>
<td>Computational experiments</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Scenarios</td>
</tr>
<tr>
<td>5.5</td>
<td>Computational results</td>
</tr>
<tr>
<td>5.5.1</td>
<td>Preprocessing</td>
</tr>
<tr>
<td>5.5.2</td>
<td>Two-way path two-way algorithm</td>
</tr>
</tbody>
</table>

6 CONCLUDING REMARKS | 113

APPENDIX | 117

CITED LITERATURE | 121

VITA | 124
## LIST OF TABLES

<table>
<thead>
<tr>
<th>TABLE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>PREPROCESSING STATISTICS</td>
</tr>
<tr>
<td>II</td>
<td>ARTIFICIALLY DECREASED UPPER BOUNDS</td>
</tr>
<tr>
<td>III</td>
<td>COMPUTATION TIME, <em>COMMUTE</em> SCENARIO</td>
</tr>
<tr>
<td>IV</td>
<td>COMPUTATION TIME, <em>NIGHT OUT</em> SCENARIO</td>
</tr>
<tr>
<td>V</td>
<td>TRIP DURATION, <em>COMMUTE</em> SCENARIOS</td>
</tr>
<tr>
<td>VI</td>
<td>TRIP DURATION, <em>NIGHT OUT</em> SCENARIOS</td>
</tr>
<tr>
<td>VII</td>
<td>PROPOSED APPROACH COMPARED TO UPPER BOUND</td>
</tr>
<tr>
<td>VIII</td>
<td>BASIC AND SIMULTANEOUS ALGORITHMS COMPARISON</td>
</tr>
<tr>
<td>IX</td>
<td>EXAMPLES FOR THE <em>COMMUTE</em> SCENARIO, BY CAR</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>14</td>
</tr>
<tr>
<td>4</td>
<td>18</td>
</tr>
<tr>
<td>5</td>
<td>20</td>
</tr>
<tr>
<td>6</td>
<td>25</td>
</tr>
<tr>
<td>7</td>
<td>29</td>
</tr>
<tr>
<td>8</td>
<td>31</td>
</tr>
<tr>
<td>9</td>
<td>35</td>
</tr>
<tr>
<td>10</td>
<td>36</td>
</tr>
<tr>
<td>11</td>
<td>44</td>
</tr>
<tr>
<td>12</td>
<td>45</td>
</tr>
<tr>
<td>13</td>
<td>48</td>
</tr>
<tr>
<td>14</td>
<td>49</td>
</tr>
<tr>
<td>15</td>
<td>50</td>
</tr>
<tr>
<td>16</td>
<td>51</td>
</tr>
<tr>
<td>17</td>
<td>55</td>
</tr>
<tr>
<td>18</td>
<td>61</td>
</tr>
<tr>
<td>19</td>
<td>68</td>
</tr>
<tr>
<td>20</td>
<td>73</td>
</tr>
<tr>
<td>21</td>
<td>74</td>
</tr>
<tr>
<td>22</td>
<td>75</td>
</tr>
<tr>
<td>23</td>
<td>75</td>
</tr>
<tr>
<td>24</td>
<td>76</td>
</tr>
<tr>
<td>25</td>
<td>77</td>
</tr>
<tr>
<td>26</td>
<td>90</td>
</tr>
<tr>
<td>27</td>
<td>95</td>
</tr>
<tr>
<td>28</td>
<td>101</td>
</tr>
<tr>
<td>29</td>
<td>102</td>
</tr>
<tr>
<td>30</td>
<td>105</td>
</tr>
<tr>
<td>31</td>
<td>105</td>
</tr>
<tr>
<td>32</td>
<td>106</td>
</tr>
<tr>
<td>33</td>
<td>106</td>
</tr>
<tr>
<td>34</td>
<td>108</td>
</tr>
<tr>
<td>35</td>
<td>109</td>
</tr>
<tr>
<td>36</td>
<td>111</td>
</tr>
<tr>
<td>37</td>
<td>111</td>
</tr>
</tbody>
</table>
LIST OF FIGURES (Continued)

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>DESCRIPTION</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>38</td>
<td>Two-way trip using 2WSA.</td>
<td>112</td>
</tr>
<tr>
<td>39</td>
<td>Optimizing legs of a two-way trip separately</td>
<td>112</td>
</tr>
</tbody>
</table>
LIST OF ABBREVIATIONS

2WMMSPP 2-Way Multi-Modal Shortest Path Problem. viii, 38

2WSA Two-Way Search Algorithm. viii, 70

ALT A*, Landmarks and Triangle inequality. viii, 17, 32, 33

ANR Access-Node Routing. viii, xii, xiv, 33–36, 52–54, 57

API Application Programming Interface. viii, 84, 92

CALT Core-based ALT. viii

EAP Earliest Arrival Problem. viii, xiii, 8, 21, 22, 42

ET-2WPNRP Effective Time Two-Way Park-aNd-Ride Problem. viii, 40–42, 44, 46, 47, 52, 84, 93, 98, 110, 113, 115

FIFO First In First Out. viii, 19, 21, 26, 27, 30, 54, 91

GTFS General Transit Feed Specification. viii, 84, 88

LCSPP Label-Constrained Shortest Path Problem. viii, 13, 14

LCSPP-D LCSPP-Dijkstra. viii, 32

LDP Latest Departure Problem. viii, 22, 23, 42

NNP Nearest Neighbor Problem. viii

OTP OpenTripPlanner. viii, xii, 83–86, 88, 92, 100, 114–116, 118, 120

PNR Park aNd Ride. viii, 54

REST REpresentational State Transfer. viii, 84

RLCSPP Regular Language-Constrained Shortest Path Problem. viii, 32, 33, 39
LIST OF ABBREVIATIONS (Continued)

**SDALT** State-Dependent ALT. viii, 33

**SPP** Shortest Path Problem. viii, xiii, 8, 21, 22

**TDCALT** Time-Dependent Core-based ALT. viii, 33

**TDSPP** Time-Dependent Shortest Path Problem. viii, 26, 27

**TNR** Transit-Node Routing. viii, 33, 34

**UCCH** User-Constrained Contraction Hierarchies. viii
With the growing phenomenon of urbanization and people moving towards urban areas, public transportation networks and road infrastructures are constantly improved and extended. Route planning within these urban scenarios is thus becoming a very important problem to address. This is also a time where mobile and real-time information are becoming more and more important. We can see this trend also in route planning: navigation systems for cars are already a commodity, smartphones and PCs are able to access different route planning services, public transit authority’s line data and much more. Some of these devices and services are confined within their domain (e.g., a car navigation system plans only road routes), others try to plan routes involving multiple modes of transportation. The last ones solve the so called multi-modal route planning problem. All of these usually work by stating a departure location and a destination, together with a departure time and the desired transportation modes for the route (including or excluding certain combinations, for example). Their output is an (sometimes sub-) optimal route that contains the roads and public transportation lines to use for the trip. The criterion they often optimize is travel time, but depending on the domain and application, others have been used too, e.g., number of mode changes or monetary cost. In some cases, more than one metric needs to be optimized (travel time and number of mode changes), making the problem a multi-criteria route planning problem.

Routing in general has been a widely researched and experimented problem, with many different combinations of characteristics explored. The scenario we specifically focus on in this
SUMMARY (Continued)

work is finding a *two-way route* between two locations using a private vehicle (like a car, a motorcycle or a bicycle) for the first part of the trip, after which it will be parked and public transportation will then be used for the remaining part (so-called *park and ride*). The parked vehicle will then have to be retrieved during the return trip, before returning to the departure location. By two-way route we thus mean a route from a departure location to a destination and from the destination back to the departure location at a later time, opposed to classic one-way routing which simply finds the best route in one direction. The goal of this work is to optimize both directions of the two-way trip together, by choosing an optimal parking location which makes the combination of the two legs optimal. This kind of route planning has not been researched as widely as classic one-way route planning, but has immediate real-world applications. In fact, when looking at the scenario where the user starts his journey with a private vehicle (a car or a bicycle), we immediately observe why it is important to search for an optimal path taking into account both legs: the user has to park the vehicle on the outward leg and needs to retrieve it on the return leg. The parking location that is chosen during the first leg is in fact a very important constraint on the second leg and it is not possible to optimize both legs separately, as it could result in having two different parking locations for the two legs. On the other hand, optimizing only one of the two legs and then finding the best route that uses the chosen parking location may lead to sub-optimal routes. Indeed, when moving from the departure location to the destination, one could choose a particular parking location which is the best choice for the outward trip, but which is not served by public transportation at the time of the return trip. In other cases the chosen parking location could force the user to use
SUMMARY (Continued)

streets which are heavily influenced by traffic during the return trip (e.g., in rush hours), thus making the return trip much longer than needed.

Solving this problem efficiently is particularly impactful in many realistic scenarios involving entering the city and then exiting it at a later time, like a person living in the suburbs that has an appointment in the city center or wanting to find the best combination of transportation modes to get to work downtown and back home. The shortest paths for the two legs of the two-way trip will in fact usually be significantly different when computed separately, since the timetables of public transportation and the traffic conditions change during the day. For example, if the user wants to return in the evening or at night, she might have to use different bus routes because those she used in the morning do not run after a certain time.

The major contributions of our work include:

- Extension of concepts from Access-Node Routing (ANR) to the Park-aNd-Ride scenario;
- Extension of concepts from ANR and the presented PNR-nodes to a time-dependent road network;
- A new approach to find two-way trips in a multi-modal urban-sized transportation graph, simultaneously optimizing the combination of the two legs of a two-way trip, that finds optimal solutions which are 20% better than their upper bounds on average;
- A simple approximation method to add simple time-dependency to a time-independent road network;
- Implementation of the presented approach as an extension of OpenTripPlanner (OTP), an open-source trip planning platform.
Our work is organized as follows.

Chapter 1: Preliminaries.

Chapter 1 lays the theoretical foundations for our work. We start by defining basic concepts of graph modeling (Section 1.1). we present models for each of the network types needed for our work.

Chapter 2: Related work.

In Chapter 2 we present relevant related works. We first describe a formal definition of the Shortest Path Problem (SPP), as well as Dijkstra’s algorithm, which can solve the Shortest Path Problem on uni-modal networks. We proceed to illustrate other more complex routing algorithms within their domain and characteristics, first for uni-modal networks and then for multi-modal networks. Finally we describe the Earliest Arrival Problem (EAP) and its relation to the SPP, as well as the resulting time-dependent routing algorithms solving it.

Chapter 3: Problem definition.

In Chapter 3 we analyze and constrain the problem we want to tackle to a set of aspects we want to focus on. We start by defining the problem and some needed concepts in Section 3.1, also extending definitions that can be found in literature to reflect the aspects we are focusing on and showing differences. In Section 3.2 we make some simplifying assumptions and specify actual constraints we want to set for the problem.
Chapter 4: Proposed approach.

In Chapter 4 we start from all the previous works illustrated in Chapter 2, in particular the research on ANR, to present an efficient and exact algorithm which solves the problem described in Chapter 3. The proposed algorithm has been created as a two-phase algorithm, with an offline, one-time preprocessing phase (Section 4.2) and an optimization phase which actually performs the search (Section 4.3).

Chapter 5: Experimental results.

Chapter 5 describes how we conducted the computational experiments supporting the theoretical solutions presented in Chapter 4 and shows their results. We start by presenting the data used as input throughout the experiments in Section 5.3, followed by a detailed description of the implementation that has been developed (extending an open source trip planning platform) in Section 5.2. In Section 5.5 we show some examples of how the developed algorithm performs and analyze these performances.

Chapter 6: Concluding Remarks.

Lastly, in Chapter 6 we sum up our contributions, presenting the conclusions that can be drawn from our work and giving an outlook of what future research can be performed on the presented topics.
CHAPTER 1

PRELIMINARIES

In this chapter we illustrate some basic theoretical concepts needed throughout the work. As all algorithms to be described later in this work on graphs, first we will introduce elementary concepts of graph theory. We will then proceed with describing transportation network modeling for different kinds of transportation networks.

1.1 Graph Theory

Graph.

A graph \( G = (V,A) \) is a tuple consisting of a finite set of nodes \( v \in V \) and a set of arcs \( (i,j) \in A \subseteq V \times V, i,j \in V \). There is an arc from a node \( u \in V \) to a node \( v \in V \) if and only if \( (u,v) \in A \).

Arrows can be either directed or undirected, depending on whether the direction of the arc is important. Given a directed arc \( (u,v) \) from \( u \) to \( v \), \( v \) is the head and \( u \) is the tail of the arc. An arc \( (u,u) \) which connects a node to itself is called a self-loop. In this work we will use graphs with directed arcs exclusively (directed graphs, opposed to undirected graphs).

The graph obtained by inverting all arcs is called the backward graph \( \overrightarrow{G} = (V, \overrightarrow{A}) \), where \( (u,v) \in \overrightarrow{A} \iff (v,u) \in A \).
A node induced subgraph $G' \subseteq G$ with $G' = (V', A')$ and $V' \subseteq V$ is obtained by the arc set $A' = \{(u, v) \mid u \in V', v \in V'\}$ and $(u, v) \in A$, while an edge induced subgraph $G' \subseteq G$ with $G' = (V', A')$ and $A' \subseteq A$ is obtained by the node set $V' = \{v \mid \exists (u, v) \in A'\}$ or $\exists (v, u) \in A'$, $u \in V$.

Furthermore, the union of two graphs $G_1 = (V_1, A_1)$ and $G_2 = (V_2, A_2)$ is defined as the graph $G_u = G_1 \cup G_2 = (V_1 \cup V_2, A_1 \cup A_2)$.

Arc costs.

Every arc has an arc cost associated to it. In route planning arc costs usually represent travel times or other metrics that need to be optimized. When representing travel times, in many cases constant arc costs are perfectly sufficient: $c_{uv}$ represents the cost for arc $(u, v) \in A$ at any time of the day. In Section 2.4 we will see how this will not be the case for other settings.

Paths.

A path $p$ in $G$ is a sequence of nodes $p = [v_1, v_2, ..., v_k]$ such that the condition $(v_i, v_{i+1}) \in A$ holds for all $1 \leq i < k$. If $v_1 = v_k$ the path $p$ is called cycle. A subpath $s \subset p$ is a path which is fully contained in $p$. The cost of a path $p$ is given by the sum of the costs of the arcs it is made of and is denoted by:

$$c(p) = \sum_{i=1}^{k-1} c_{v_i, v_{i+1}} \quad (1.1)$$

The cost of the shortest path between a node $u$ and a node $v$ is denoted by $d(u, v)$. In the case of time-dependent graphs, the cost $c(p, \tau)$ of a path $p$ departing from $v_1$ at time $\tau$ is given by recursion:

$$c((v_1, v_2), \tau) = c_{v_1v_2}(\tau) \quad (1.2)$$
and then:

\[ c((v_1, ..., v_j), \tau) = c((v_1, ..., v_{j-1}), \tau) + c_{v_{j-1}, v_j}(c((v_1, ..., v_{j-1}), \tau) + \tau) \] (1.3)

### 1.2 Network Modeling

In this section we first describe graphs for single network types: walking, private bicycle, private car and public transportation. Then we describe how they can be combined to build a model of a multi-modal transportation network.

#### 1.2.1 Uni-modal networks

**Walking road network**

The model we use for the foot network is very simple: junctions for walk segments are represented by nodes and arcs between two nodes exist if there is a footpath between two junctions. Arc costs depend on the geographical distance between the two junctions and on the average pedestrian walking speed. The graph is directed, so a street segment is actually split into two arcs, one for each direction.

**Bicycle road network**

The bicycle network is similar to the above described walking network. Nodes still represent road junctions and arcs exist whenever biking is allowed between two nodes. Also, arc costs are the cycling time associated with the geographical distance between the two nodes and depend on average cycling speed.
Car road network

Analogously, when modeling a car network, nodes represent road junctions and arcs represent roads connecting them. There are different type of roads with different speed limits, so each road arc will store information about the road type. Also, arc costs can vary during the day, since travel times may depend on traffic conditions in addition to geographical distance and road type. To model one-way streets, it is sufficient to add only one arc for the desired direction.
Public transportation network

While modelling networks related to the road network like the ones described above is quite straightforward, to model public transportation networks we need to use one of the models described in the previous section about time modeling. The choice we made in this work has been to use the time-dependent realistic model, which combines lower memory requirements and good simplicity.

1.2.2 Multi-modal networks

Searching a multi-modal path involves multiple networks types at the same time, thus we need to combine the previously mentioned graphs into a single multi-modal network. To combine all of the single-mode networks (the road graph $G_{\text{road}}$ and the public transportation graph $G_{\text{transit}}$) into one unified multi-modal network, we need to insert link arcs between nodes belonging to different networks which are geographically near to each other. To find which node has to be linked to which other node, the Nearest Neighbor Problem needs to be solved.

The Nearest Neighbor Problem

The Nearest Neighbor Problem consists in finding the nearest candidate point $p$ in a set of points $P$ for each search point $q \in Q$ with respect to some metric $d$.

Let $\mathbb{R}^n$ be an $n$-dimensional vector space over $\mathbb{R}$ and $P \in \mathbb{R}^n$ a finite set of vectors called candidate points. Also let $d : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ be a metric on $\mathbb{R}^n$. Then the Nearest Neighbor Problem is defined \cite{1}\cite{2} as follows.
Definition 1.1 (Nearest Neighbor Problem). Given a metric space \((\mathbb{R}^n, d)\), a set of candidate points \(P\) on \(\mathbb{R}^n\) and a set \(Q\) of search points on \(\mathbb{R}^n\), we ask for a map \(f : Q \rightarrow P\) with the property:

\[ f(q) = p \iff \forall p' \in P : p \neq p' \implies d(p', q) \geq d(p, q) \]  \hspace{1cm} (1.4)

Two approaches are described in [1] to solve this problem, a linear search approach and one based on \(k\)-d-trees. The first one is a more naive solution, taking each search point \(q \in Q\) and scanning the list of candidate points \(P\) for the point having minimum distance to \(q\). This solution can be easily implemented but requires a running time of \(O(|Q| \cdot |P|)\), which is not feasible for large sets \(P\) and \(Q\). Thus the need for a more efficient solution.

Bentley [3] introduced \(k\)-d-trees, a data structure built on the idea to generalize a binary search tree to \(k\) dimensions. This approach achieves an average logarithmic time for search
searches. Since each search can be answered in average time \( O(\log |P|) \), the running time for all the searches becomes \( O(|Q| \log |P|) \).

**Merging the graphs**

The merge operation is performed in two steps: *merging* and *linking*. During the merging phase, starting from all uni-modal graphs \( G_1, G_2, ..., G_k \) a single multi-modal graph \( G = (V, A) \) where the node and arc sets are simply the union of the node and arc sets of the uni-modal graphs: \( V = V_1 \cup V_2 \cup ... \cup V_k \) and \( A = A_1 \cup A_2 \cup ... \cup A_k \). To distinguish between different types of nodes, each of them is labeled depending on the original graph they came from (e.g., `PARKING NODE`, `ROAD NODE`, `TRANSIT STOP` ..). During the linking phase, link arcs are inserted between the different networks within the multi-modal graph \( G \). Since one execution of the linking operation links only two graphs together, the operation has to be performed multiple times.
CHAPTER 2

LITERATURE REVIEW

In this chapter summarize important parts of the research done on related problems, from which we have been inspired and upon which we built our work. We will first describe the Shortest Path Problem (SPP) in Section 2.1 and its fundamental importance in routing, together with its most important variations. In Section 2.4 we discuss the importance of time-dependency in routing scenarios and how arc costs need to be changed in order to account for the time-dependency. We proceed by defining the Earliest Arrival Problem (EAP) and its relation with the SPP.

2.1 The Shortest Path Problem

The SPP is one of the fundamental problems in algorithm and operations research. It is defined as follows.

**Definition 2.1 (Shortest Path Problem).** Given a weighted, directed time-independent, time-dependent or mixed graph $G = (V, A)$, a source node $s \in V$, a target node $t \in V$ and a departure time $\tau < \Pi$, we ask for a path $p = [v_1, ..., v_k]$ with the following properties:

1. The path begins at $s$, thus $v_1 = s$,
2. the path ends at $t$, thus $v_k = t$,
3. for all paths $p'$ having the above properties it has to hold that $c(p', \tau) \geq c(p, \tau)$.

If $G$ is time-independent, $\tau$ is ignored and the cost of a path $p$ is simply $c(p)$.
Obviously, different variations of this problem exist:

- **Many-To-Many Shortest Path Problem**: instead of having only one source node and one target node, we have a set of source nodes $S \subseteq V$ and a set of target nodes $T \subseteq V$. We thus want to find a shortest path for every pair $(s, t) \in S \times T$ of source and target nodes.

- **One-To-All Shortest Path Problem**, which is a special case of the above Many-To-Many Shortest Path Problem: the set of source nodes $S$ contains only one node and the set of target nodes is the set of all nodes $T = V$. Thus we are seeking the shortest paths from $s \in S = s$ to every node $v \in V$. Since the arc set of the resulting paths forms a tree rooted in $s$, we are thus computing a *shortest path tree*.

- **All-Pairs Shortest Path Problem**: a special case of the Many-To-Many Shortest Path Problem, where both $S = V$ and $T = V$, *i.e.*, we are computing distances among all pairs of nodes in the graph. This is obviously very expensive with respect to execution time and memory consumption.

As we will see in the next section, Dijkstra’s algorithm [4] can solve all these problems with only minor modifications.

**Time modeling**

When considering time in graphs, we can classify graphs in two categories: *time-independent* graphs and *time-dependent* graphs. The main difference between these categories is due to how the arc costs are assigned: they are constant in the first case and depend on the time at which
they are evaluated in the other. We can show the difference with two very simple examples. When modeling a simplified walking graph where walking the same street takes always the same time, a time-independent graph can be used. On the other hand, if we want to model a road network to be traversed by car, traffic often delays trips (think of the same road at rush hour or at night).

2.2 Time-Independent Routing

Time-independent routing has been studied extensively in many previous works (e.g., [5]) and has been often used to model simplified road networks for both walking and car traversal. In this work, we will use a time-independent graph to model the walking and bicycle road networks. In a time-independent graph $G = (V, A)$ each arc $(i, j) \in A$ is assigned one constant cost $c_{ij}$ which usually represents travel time or geographical distance, but can be any other metric one wants to optimize. Not having time-dependencies makes the model much simpler, although in many cases less realistic. To solve shortest path searches on such a graph we can use Dijkstra’s algorithm [4].

A great amount of research has been focusing on accelerating shortest path searches on time-independent graphs, resulting in great improvements and speed-up techniques. Very detailed overviews of this research can be found in [5] [6] and [7]. While in some cases a time-independent model can be used effectively without oversimplifying the problem, in other cases time-dependency is inherent to the problem. It is the case of public transportation, for example, as trains and buses only operate at certain times and therefore the choice of the shortest path highly depends on the departure time of the trip. In the next section we will see how we can
deal with time-dependency, either by eliminating it from the model or by incorporating it in
the model itself.

2.2.1 Uni-modal routing

When routing in a time-independent scenario, we start from a directed uni-modal graph
\( G \) with time-independent arc costs \( c_{ij} \) for every \( (i,j) \in A \). To solve the Shortest Path
Problem in this scenario, Dijkstra’s original algorithm is the basis of many other approaches.
Its pseudocode can be seen in Algorithm 1.

Although the original algorithm does not use a priority queue, later improvements proposed
the use of one to speed up running time from \( \mathcal{O}(|V|^2) \) to \( \mathcal{O}((|V| + |A|) \log |V|) \) using a binary
heap [8], and to \( \mathcal{O}(|V| \log |V| + |A|) \) using a Fibonacci heap [9] (and even better using other
modified versions). The queue is populated with nodes in the order they need to be explored,
\textit{i.e.}, ordered by tentative distance from the source node \( s \). It starts with an empty priority
queue and only the source node is inserted with a key equal to 0. Then at each iteration the
node with the lowest key is extracted. For every outward arc from the current node \( v \), its head
is inserted in the queue with a key equal to \( \text{key}(v) + c_{vw} \), where \( \text{key}(v) \) is the extracted node’s
key and \( c_{vw} \) is the arc cost. If the node \( w \) to be inserted in the queue is already present, either
its key is updated (if its key in the queue is greater than the new one), or no action is performed
(if the key is lower or equal than the new one). If the node to be inserted in the queue has
already been evaluated, it is simply skipped. When all neighbors of the current node have been
added to the queue or discarded, the node is said to have been \textit{settled} and is marked as such.
Algorithm 1 Dijkstra’s algorithm

Q ← priority queue
Q.insert(s, 0)
settled_targets ← ∅

while not Q.isEmpty() do
    v ← Q.dequeue()
    if v ∈ T then
        settled_targets ← settled_targets ∪ {v}
        if settled_targets = T then
            stop;
        end if
    end if
    for all outward arcs (v,w) do
        if not Q.contains(w) then
            Q.insert(w, dist(s,v) +c_{vw})
            pre(w) ← v
        else
            if dist(s,v)+c_{vw} < dist(s,w) then
                Q.decreaseKey(w, dist(s,v)+c_{vw})
                pre(w) ← v
            end if
        end if
    end for
end while
When all target nodes have been settled, all shortest paths to them are guaranteed to have been found and the algorithm can stop.

We can see how this algorithm can solve all variants of the Shortest Path Problem presented in the previous section, with small modifications. In fact, for cases where we have a single source node we simply stop when all target nodes have been settled (or just one, if we look for the shortest path to one of the target nodes), while for cases with multiple source nodes we execute the algorithm multiple times, once for each source node. The Many-To-Many Shortest Path Problem where we are interested in only one shortest path from $S$ to $T$ is treated differently, by adding all nodes $s \in S$ to the priority queue with key 0 at the beginning.

**2.2.2 Multi-modal routing**

In the case of multi-modal route planning, the problem of finding the quickest route is complicated by the fact that there can be a great number of combinations and sequences of different transportation modes that can be used. Routing algorithms thus need to optimize the choice of these transportation modes in a reasonable way. There are three main approaches to tackle the multi-modal routing problem. One uses a combined cost function of travel time, augmented with penalties for mode transfers. Another is to solve the Label-Constrained Shortest Path Problem (LCSPP) in order to include or exclude specific sequences and combinations of transportation modes explicitly. The last approach consists in finding Pareto sets of multi-modal routes that optimize multiple criteria in order to obtain different alternative routes.
The most interesting approach is probably the second one, which solves the multi-modal routing problem by tackling the LCSPP, i.e., by computing paths that explicitly obey constraints on the modes of transportation used. The LCSPP is tractable and solvable in deterministic polynomial time for regular languages [10], which are sufficient for reasonable scenarios in multi-modal routing.

The first algorithm to solve the problem is a modified Dijkstra’s algorithm operating on a product network of the input graph and input finite state automaton accepting the regular language provided. The product network consists of product-nodes \((v, q)\) where \(v \in V\) is a node of the original network and \(q \in Q\) is a state of the automaton. An edge exists between two product-nodes \((v_1, q_1), (v_2, q_2)\) in the product network if and only if there is an edge in the original graph connecting \(v_1\) to \(v_2\) and there is a label for which there is a transition from \(q_1\) to \(q_2\) in the automaton. The resulting product graph is thus unimodal, and a shortest path
can be found by using an arbitrary algorithm to solve the Many-To-Many-Shortest Path Problem where the source nodes are all product-nodes \((v, q)\) where \(v = s\) and the target nodes are all product-nodes \((v, q)\) where \(v = t\).

**Algorithm 2** Time-Independent Multi-Modal Dijkstra’s algorithm

1. \(Q \leftarrow \text{priority queue}\)
2. \(\text{for all } q_s \in S \text{ do}\)
   - \(Q.\text{insert}(s, 0)\)
3. \(\text{end for}\)
4. \(\text{settled_targets} \leftarrow \emptyset\)
5. \(\text{while not } Q.\text{isEmpty()} \text{ do}\)
   - \((v, q) \leftarrow Q.\text{dequeue()}\)
   - \(\text{if } v \in T \text{ and } q \in F \text{ then}\)
     - \(\text{settled_targets} \leftarrow \text{settled_targets} \cup \{v\}\)
     - \(\text{if settled_targets} = T \text{ then}\)
       - \(\text{stop};\)
   - \(\text{end if}\)
   - \(\text{for all outward arcs } (v, w) \text{ do}\)
     - \(\text{for all states } q' \in \delta(q, label(v, w)) \text{ do}\)
       - \(\text{if not } Q.\text{contains}((w, q')) \text{ then}\)
         - \(Q.\text{insert}((w, q'), \text{dist}(s,(v,q)) + c_{vw})\)
         - \(\text{pre}(w) \leftarrow v\)
       - \(\text{else}\)
         - \(\text{if dist}(s,(v,q)) + c_{vw} < \text{dist}(s,(w,q')) \text{ then}\)
           - \(Q.\text{decreaseKey}((w, q'), \text{dist}(s,(v,q)) + c_{vw})\)
           - \(\text{pre}(w) \leftarrow v\)
       - \(\text{end if}\)
     - \(\text{end if}\)
   - \(\text{end for}\)
   - \(\text{end if}\)
- \(\text{end while}\)
2.3 Speed-up Techniques

We now briefly present some of the most important speed-up techniques research has brought forward to increase performance of Dijkstra’s algorithm, mainly by reducing its search space. Broader and more comprehensive overviews of search techniques can be found in [5], [6] and [7]. The main ingredients for unimodal route planning common to many modern speed-up techniques are: Bidirectional search, Contraction, Goal-oriented search.

Bidirectional search simultaneously computes the shortest path from the source $s$ to the target $t$ and the shortest path from the target to the source on the backward graph, stopping when the two searches meet in the middle. The actual shortest path is then found by combining the partial paths found by the two searches. Bidirectional search decreases running time and memory requirements, since the search space is vastly reduced by using two “smaller” searches.

Goal-oriented search algorithms try to guide the search toward the target, in order to avoid exploring nodes that are not in the direction of the target and thus reducing search space to nodes that might be relevant for a shortest path. The most important approaches are based on the A* search algorithm, which uses a lower bound on the distance $\text{dist}(v, t)$ as a potential function $\pi : V \rightarrow \mathbb{R}$ for every node $v \in V$. This potential function is then used in a modified Dijkstra’s algorithm where the key of the priority queue is set to $\text{dist}(s, v) + \pi(v)$ (instead of $\text{dist}(s, v)$), which itself is a lower bound on the shortest path from $s$ to $t$ that uses $v$. By doing so, vertices that are closer to the target are explored earlier in the search. In a particular case, when $\pi$ is an exact lower bound (i.e., $\pi(v) = \text{dist}(v, t)$, $\forall v \in V$), only those vertices which are on the shortest $s - t$ paths would be explored and the search space would be minimal. To
produce a correct shortest path, the potential function has to be feasible, i.e., it has to hold that \( \text{dist}(v, u) \geq \pi(v) - \pi(u) \). A later improvement to \( A^* \) has been to compute feasible potential functions using the triangle inequality. This approach is called \( A^*, \) Landmarks and Triangle inequality (ALT) \([11]\). During preprocessing a small set of landmarks is picked and distances between each landmark and all other vertices are found and stored for the optimization phase. These are then used together with the triangle inequality to compute valid lower bounds for the \( A^* \) search algorithm.

Hierarchical speed-up techniques try to exploit the inherent hierarchy of road networks, i.e., the fact that shortest paths eventually converge to a limited set of important roads, if the paths are long enough. Contraction Hierarchies \([12]\) is an approach which adds shortcuts to the road network to contract the graph and which then uses the contracted graph for long shortest path searches, thus skipping “unimportant” road vertices. The algorithm works in two steps: a preprocessing phase where nodes are ordered by importance and contracted from least to most important, and a search phase where a bidirectional search is performed on the graph \( G \) augmented by the shortcuts computed in the previous stage. During the optimization phase only arcs leading to nodes of higher importance are visited.

A somewhat similar approach, though not using contraction and used for public transportation, is Transit-Node Routing \([13][14]\) where a set of relevant transit nodes is determined where almost all “far” shortest paths pass through. Distances between each pair of these transit nodes are then precomputed. From the set of transit nodes, for each vertex \( u \in V \setminus T \), a set of access nodes is chosen, i.e., nodes which are the first transit node contained in a shortest path from \( u \).
In addition to the access nodes, preprocessing also stores the distances from $u$ to each access node. The search algorithm optimizes the total $s - a(s) - a(t) - t$ distance, where $a(s)$ and $a(t)$ are the access nodes for source $s$ and target $t$.

To solve contraction’s issues in the multi-modal case, in the same work Pajor presents **Core-Based Routing**, which is purely multi-modal and based on contraction. This approach restricts preprocessing and contraction to the road network, which is assumed to be time-independent, and thus avoids the disadvantages of time-dependent contraction. This approach also allows to use bidirectional search on the component graph (the non-core part of the graph), since it is time-independent. Routing on the core is performed with a simple unidirectional time-dependent multi-modal **Dijkstra**, but the algorithm to be used on the core can be chosen orthogonally since it is conceptually “detached” from the component graph.

![Figure 4: Core-based routing.](image)

Entry (blue) and exit (red) nodes to the core part of the graph (grey on the left, green on the right) are stored for every node, and a path between those is found in the core graph (pink).
2.4 Time-Dependent Routing

Time-dependent shortest path search leads to a great amount of new problems and characteristics having increasing importance. It starts from the arc costs of the graph. We described in Section 2.2 how for time-independent graphs constant values are sufficient to model the arc costs. For time-dependent route planning, we need to generalize the concept of arc costs, since we want to be able to have different values assigned to the same arc for different times of the day. This can be achieved by having cost functions instead of constant costs for those arcs. All cost functions are elements of a function space \( \mathbb{F} \) consisting of functions \( f : \mathbb{R}_{0}^{+} \rightarrow \mathbb{R}_{0}^{+} \). The cost functions associated to an arc \((i, j) \in A\) are given by \( c : A \rightarrow \mathbb{F} \) and are denoted by \( c_{ij}(\tau) \).

In this work, we restrict ourselves to periodic cost functions with a period \( \Pi \). This means that \( \forall \tau \geq \Pi, \ c_{ij}(\tau) = c_{ij}(\tau \mod \Pi), \ i.e., \ c_{ij}(\tau + k\Pi) = c_{ij}(\tau), \ \forall \tau \in \mathcal{T} = [0, \Pi], \ \forall k \in \mathbb{Z}. \)

Since costs are travel times, we additionally require that the functions fulfill the so-called First In First Out (FIFO) property, i.e., for any two time instants \( \tau_1, \tau_2 \in \mathbb{R}_{0}^{+} \) with \( \tau_1 \leq \tau_2 \) it must hold that

\[
c(\tau_1) + \tau_1 \leq c(\tau_2) + \tau_2, \ \forall c \in \mathbb{F}
\]

(2.1)

The FIFO property is sometimes also referred to as the non-overtaking property, since it ensures that along an arc it is never possible to overtake, i.e., to depart later but arrive earlier.

For every cost function, the lower and upper bounds of \( c_{ij}(\tau) \) are defined as \( \underline{c}_{ij} = \min_{\tau \in \mathcal{T}} c_{ij}(\tau) \) and \( \overline{c}_{ij} = \max_{\tau \in \mathcal{T}} c_{ij}(\tau) \), respectively. The graphs using the lower bounds or upper bound of
Figure 5: **Examples of cost functions.** Public transportation has a piecewise linear cost which is decreasing up until the bus/tram departs, after which it increases immediately to a new maximum which accounts for waiting at the stop. On the other hand, the road network’s arc cost may increase and decrease depending on traffic, thus they might break the FIFO property if the slope of the graph is too steep.

the cost functions as arc costs are marked by $G$ and $\mathcal{G}$, respectively. Obviously, the type of cost function which is used influences the efficiency of the search algorithms applied on the graph.

Dean[15] presented an extensive theoretical study of different cost functions for different algorithms. We will use piecewise linear cost functions throughout this work, since they are a natural fit for timetable information and they can be used also to model traffic information. But their most important feature is their simplicity and efficiency in computational implementations. In fact, since a piecewise linear function is composed of straight-line parts, it can be modeled by a finite set $\mathcal{B}$ of interpolation points. Each of those interpolation points $b_i \in \mathcal{B}$ consists of a departure time $\tau_i$ and an associated cost function value $c_{ij}(\tau_i)$. For every departure time $\tau_j$ which is not in $\mathcal{B}$, the function value $c_{ij}(\tau_j)$ can be computed by interpolation. It is also very
easy to check if a piecewise linear cost function satisfies the above mentioned FIFO property, since the condition
\[ c_{ij}(\tau_1) + \tau_1 \leq c_{ij}(\tau_2) + \tau_2, \forall \tau_1 \leq \tau_2 \]  
(2.2)
can be reformulated as \( c_{ij}(\tau_2) - c_{ij}(\tau_1) \geq -(\tau_2 - \tau_1) \), i.e.,
\[ \frac{dc_{ij}}{d\tau} \geq -1 \]  
(2.3)
This means we only have to evaluate the slope of the linear pieces to see if the arcs fulfill the property.

**Earliest Arrival Problem**

Time-dependent routing make it possible to address new problems which could not be solved by using a time-independent model. For example, when planning a trip from a starting location to a destination, the first metric we think of to optimize is travel time. This usually means either minimizing the actual time the trip takes or minimizing the arrival time, since departing later might give a shorter path which actually arrives later than a longer path departing earlier. Actual travel time is minimized by solving the SPP defined in Section 2.1. The EAP tackles the problem of optimizing arrival time and is defined as follows.

**Definition 2.2 (Earliest Arrival Problem (EAP)).** Given a time-independent or time-dependent network, a source node \( s \) and a target node \( t \) in the network, as well as a departure time \( \tau < \Pi \), we ask for a path in the network with the following properties:

1. The path must start at \( s \),
2. the departure time at $s$ is at least $\tau$,

3. the path ends at $t$,

4. the arrival time of all other routes satisfying the above properties must be greater or equal.

The second demand can be dropped if the network is entirely time-independent, since the arc costs are constants and the departure time does not influence the choice of the shortest path. Thus one can immediately see how the two problems are related. In fact, in the case of entirely time-independent graphs, solving the EAP is equivalent to solving the Shortest Path Problem (SPP), as the path arriving earliest at the destination is also the shortest path (which is not dependent on the departure time, and thus is the same for every departure time).

A similar definition can be given for the Latest Departure Problem (LDP), where we give a maximum arrival time at the target $t$ and we want to find the path that maximises the departure time. The main difference between these two problems (LDP and EAP) and the SPP lies in what function is being optimized: the SPP minimizes the actual cost of the path, regardless of waiting at the departure or arrival location, while the LDP and EAP also take into account the time spent at the source or target. To clarify, if $\tau_d$ is the actual departure time from the source, $\tau_a$ is the actual arrival time at the target, $\tau_{\text{maxArr}}$ is the maximum arrival at the target for the LDP and $\tau_{\text{minDep}}$ is the minimum departure time from the source for the EAP, the three problems optimize the following.

- the SPP optimizes $\tau_a - \tau_d$
- the EAP optimizes $\tau_a - \tau_{\text{minDep}}$
• the LDP optimizes $\tau_{\text{maxArr}} - \tau_d$

As we stated before, in a time-independent scenario these objective functions are always equal to each other, but in a time-dependent scenario this does not hold, as, e.g., using the shortest path does not necessarily mean we will arrive earliest at the destination: the departure of the shortest path might be ten minutes after our minimum departure time, while a slightly longer path might depart immediately, thus arriving earlier.

**Time modeling**

Two main approaches have been proposed to model time-dependency in graphs: the *time-expanded* approach [16][17][18] and the *time-dependent* approach [19][20][21][22]. The common feature is that the search is answered by searching for a shortest path algorithm on a suitably constructed graphs. The foundation for both models is a *timetable* from which to retrieve the data to construct the graph. A tuple $(C, B, Z, \Pi)$, where $B$ is the set of public transportation stations, $Z$ is the set of trains/buses/etc., $\Pi$ is the time period of the timetable and $C$ is the set of so-called *elementary connections*, is called a timetable. A tuple $c = (Z, S_1, S_2, \tau_1, \tau_2)$ is an elementary connection, representing train $Z \in Z$ departing station $S_1 \in B$ at time $\tau_1 < \Pi$ and arriving at station $S_2 \in B$ at time $\tau_2 < \Pi$ without intermediate stops. Thus a train route will consist of a set of elementary connections for each train that serves that route. Since the train
can depart in one period of the timetable and arrive in the next (e.g., depart before midnight and arrive after midnight), the travel time can be obtained as

\[
\Delta(\tau_1, \tau_2) = \begin{cases} 
\tau_2 - \tau_1 & \text{if } \tau_2 \geq \tau_1 \\
\Pi - \tau_1 + \tau_2 & \text{otherwise}
\end{cases}
\]

(2.4)

**Time-expanded approach**

The time-expanded approach uses the principle of time-expansion: using discrete time values, a copy of a node is maintained for every time value in the time-expanded graph (in general, although obviously not every time value is needed and thus some unnecessary copies are not actually maintained). Construction for an example railway network works as follows. A node is created for every departure and arrival time event at each station and two arc types are used. For every elementary train connection from station \(S_1\) to station \(S_2\) there is a train-arc from a departure-node of \(S_1\) associated to a departure time \(\tau_d\) to an arrival-node of \(S_2\) associated to an arrival time \(\tau_a\), as in Figure 6a. For each station node \(S\), all nodes associated to \(S\) are kept in order according to their time value. Moreover, stay-arcs \((v_i, v_{i+1})\), \(1 \leq i \leq k-1\) and \((v_k, v_1)\) are used to connect the time events of \(S\) during \(\Pi\), which represent waiting at \(S\).

To find a shortest path on the time-expanded graph, one can use almost any kind of search algorithm working on graphs, as arc costs are actually constant and only slight modifications are needed to adapt them to the time-expanded graph. For example, Dijkstra’s algorithm [4] can be applied with very limited effort. The major issue of this approach is obviously the size of the resulting graph, which in the case of a major city’s public transportation network with
Figure 6: **Time-expanded station model.** The simple model connects departure nodes (d) of one station to arrival nodes (a) of the following station. The realistic model also accounts for transfer times, using transfer nodes (t).

many trips of the same transit line during the day would make the graph difficult to even store explicitly. Moreover, with the time-expanded approach, it is also impractical to model time-dependent road networks, since the presence of a timetable is mandatory. In fact, departures and arrivals by car are not scheduled at fixed times but can happen at any time instant. The time-expanded model of a time-dependent road network would thus need to have a number of time events equal to the number of time units during the day (usually a second, thus 86400). This clearly is not a viable option, if not for timetable scheduled networks.
Time-dependent approach

The time-dependent approach first, investigated by Orda and Rom [19] and then further researched in works like [22], is also based on a directed graph. The difference lies in the fact that it has only a single node for every station, and there is an arc \((i, j)\) from the station \(S_1\) to the station \(S_2\) if there is at least one connection from \(S_1\) to \(S_2\). The cost of the arc depends on the time at which the arc will be used by the search algorithm applied on the graph. The cost of the arc is thus a function and not a constant value. Thus, if the \(T\) denotes the set of time values, then the cost of an arc \((u, v)\) is given by \(c_{uv}(\tau) = f_e(\tau) - \tau\), where \(\tau\) is the departure time from node \(u\), \(f_e : T \rightarrow T\) is a function such that \(f_e(t) = \tau'\) and such that \(\tau' \geq \tau\) is the earliest arrival time possible at \(v\), given the departure time \(\tau\). The cost function \(c_{uv}(\tau)\) can be viewed as a delay function, since waiting time at node \(u\) can be incorporated into the arc cost. In some cases, waiting at node \(u\) to use the same arc \((u, v)\) pays off, but in the case of First In First Out (FIFO) delay functions waiting never pays off since the earliest arrival time at \(v\) is the one associated to the earliest departure time from \(u\). Thus the FIFO assumption keeps the level of complexity acceptable: the Time-Dependent Shortest Path Problem (TDSPP) is polynomially solvable when applied on FIFO networks [23], while it is NP-hard when applied on non-FIFO networks [19]. Also, the inverse problem of the Time-Dependent Shortest Path Problem (TDSPP) (i.e., to find the latest departure time from the source node with
a given arrival time at the target) can be reduced to a TDSPP. A function $c_{uv}(\tau)$ is FIFO if it satisfies the condition:

$$\tau_1 \leq \tau_2 \implies \tau_1 + c_{uv}(\tau_1) \leq \tau_2 + c_{uv}(\tau_2) \quad (2.5)$$

An alternative condition uses the earliest arrival function $f_e(\tau)$ defined earlier, which has to be non-decreasing:

$$\tau \leq \tau' \implies f_e(\tau) \leq f_e(\tau') \quad (2.6)$$

If an arc $(v, w)$ has a FIFO delay function, the earliest arrival time at $w$ $\tau_w$ can be found by simply evaluating $c_{vw}(\tau_v)$, where $\tau_v$ is the arrival time at $v$.

The FIFO assumption is a reasonable assumption, as we will see later in this work. Thus, assuming that the FIFO property holds on all arcs of the time-dependent graph, a modified version of Dijkstra’s algorithm can be used to find the shortest path between two nodes. The differences with respect to the original Dijkstra’s algorithm are that the distance label $\delta(s)$ of the starting node $s$ is not set to 0 but to a departure time $\tau_0$ and that the arc costs are computed on-the-fly when relaxing nodes. Arc costs are evaluated as follows. Whenever an arc $(v, w)$ is considered, the distance label $\delta(v)$ of node $v$ from $s$ is known to be optimal and $\delta(v)$ is the earliest arrival time at $v$. Thus the earliest arrival time at $w$ from $v$ using arc $(v, w)$ is the arrival time associated to the earliest departure time from $v$ using the arc, $f_e(\delta(v))$, and the arc cost can thus be computed easily and is given by $c_{vw}(\delta(v))$. 
Realistic timetable information model

In [24][25] the time-dependent model is extended for the case of a railway network using information on the routes that trains may follow in the network. Starting from the set of train routes and their time schedules, the construction of a train-route graph is explained, both with constant and variable transfer cost at stations. This approach, although presented with a railway network in mind, can obviously be applied also to an urban public transportation network of buses, trams and subways. The train-route graph consists of a set of station-nodes, one for each actual train station \( S \in B \), each of which contains a set of route-nodes representing the different routes stopping at that station. The node set of the graph is the union of the set of station-nodes and of all the sets of route-nodes of all stations. Instead of connecting the station nodes directly as in the simple model explained before, in the realistic model the route nodes of subsequent stations on a train route are connected. Route nodes are connected to their station nodes as well for transfers among routes. More formally, the set of trains \( Z \) is divided into train routes \( R \), each of which contains only trains that follow the exact same sequence of stations \( [S_1, S_2, ..., S_k] \). For each station \( S_i \in R \) belonging to a train route \( R \in R \) a route node \( r_i \) is inserted into the graph and connected to \( r_{i+1} \) and to the station node \( S_i \). The arcs connecting \( r_i \) to \( S_i \) and vice versa are called transfer arcs. To allow having different transfer times for transfers among different couples of route nodes, arcs between two route nodes at the same station can be introduced. This increases the complexity of the graph regarding the number of arcs, and even if station nodes are technically not needed anymore([26]) to represent a route, they are still needed to connect the public transportation graph to the road network.
or other networks, thus adding those arcs only increases complexity. The arc set of the graph consists of four types of arcs:

- **Transfer arcs**, which can be divided into:
  - Arrival arcs, from a route-node to a station-node
  - Departure arcs, from a station-node to a route-node
  - Route change arcs, from one route-node to another (only in the case of variable transfer cost at the station)

- **Route arcs**, from a route-node at one station to the route-node of the same route at the next station of the route

Figure 7: Realistic timetable information model.
This approach obviously respects the FIFO property, since each arc in the graph represents either a part of a route or a transfer with constant cost, and trains on the same route are assumed to have the same speed and to not overtake each other (else one would be a faster train and be represented by a different route).

### 2.4.1 Uni-modal routing

To cope with time-dependency, Dijkstra’s algorithm has been adapted in two ways, depending on the type of search to be performed. In fact, when looking for shortest path in time-dependent networks, one usually looks for two types of information: either the shortest path for a fixed departure time $\tau$ (called *time routing* in [1]) or the shortest paths for all times within a specified time window (so-called *profile routing*).

**Shortest path for fixed departure times (time routing).**

To perform time routing, only minimal modifications need to be made to the time-independent Dijkstra’s algorithm, provided the arc cost functions of the graph satisfy the FIFO property. The most intuitive one is that we need to provide a departure time $\tau$ from the source node as an additional input. The second, is that we need to consider the time at which we encounter an arc when evaluating its cost. Let $(i, j)$ be an arc of which we need to compute the cost. Then the time at which we evaluate the cost function $c_{ij}(\tau)$ is the departure time $\tau$ plus the distance of $i$ from the source node $s$, $\text{dist}(s, i)$. Thus $c_{ij} = c_{ij}(\tau + \text{dist}(s, i))$. 
Shortest path for time windows (profile routing).

In profile routing, the result of the search is a piecewise linear function where each interpolation point represents a shortest path for a particular time. This kind of routing has been used in some parts of our work where the distances at all times of the day are needed.

![Figure 8: Upper and lower bounds during a time window of 24 hours.](image)

The first steps to solve a profile search have been presented in [15], according to which a profile search from $s$ to $t$ can be computed as follows. Let $dist^*(s, v)$ denote the profile function that contains the distance of $v$ from $s$ at all times of the day. DIJKSTRA’s algorithms is then modified by using $dist^*(s, v)$ as distance label for the nodes. As the key of the priority queue
the lower bound of the respective distance label is used. Since it is possible that a node is inserted more than once into the queue, the algorithm can stop only when the lowest key in the queue is larger than the upper bound of the distance function associate to the target. If this is the case, the shortest path to $t$ has been found.

2.4.2 Multi-modal routing

As we described in Section 2.4.2, the first algorithm to solve the Regular Language-Constrained Shortest Path Problem (RLCSPP) is a modified Dijkstra’s algorithm operating on a product network of the input graph and input finite state automaton accepting the regular language provided. Pajor [1] adapts this modified version of Dijkstra’s algorithm, called LCSPP-Dijkstra (LCSPP-D), to time-dependency. In the same work, the author presents modifications of some of the speed-up techniques presented in Section 2.3 (e.g., bidirectional search, A*, Landmarks and Triangle inequality (ALT), Arc-Flags and shortcuts) to both multi-modality and time-dependency. The result of his work clearly shows that some techniques (ALT, contraction) are more easily augmented to both time-dependency and multimodality than others. Bidirectional search can be adapted easily to multi-modality but needs significant modifications to work on time-dependent graphs, since the arrival time at the destination is not known in advance. On the opposite side there is Arc-Flags, which can be adapted to time-dependency with only an increase in preprocessing times, but which is also very hard to generalize to multi-modal networks.
Time-Dependent Core-ALT

From Core-Based Routing and from unimodal Time-Dependent Core-based ALT (TD-CALT) [27] Pajor[1] also derives a multi-modal version of Time-Dependent Core-ALT, in order not to have the complexity of the bidirectional time-dependent algorithm, to be used on the time-dependent core. Bidirectional TDCALT actually slows down the search with respect to the unimodal algorithm, because of its complexity and the bad lower bounds for the backward search.

State-Dependent ALT

Kirchler[2] introduces a new algorithm, State-Dependent ALT (SDALT), which solves the RLCSPP on a multi-modal network. SDALT is an adaptation of the speed-up technique ALT which performs significantly better than the modified Dijkstra algorithm to solve the RLCSPP. SDALT uses an automaton accepting a predefined regular language to precompute state-dependent distances, providing lower bounds per vertex and state. Thus, like Access-Node Routing, SDALT does not handle search time specification of the regular language constraints.

2.4.3 Access-Node Routing

In [1], Pajor presents ANR (also partly presented in [28]), taking some ideas from Transit-Node Routing (TNR)[13] and restricting the use of the road network to the beginning and end of the path. This work is be the basis on which we built our contribution. The key idea of ANR is to precompute sets of relevant (entry- and exit-) access-nodes for the public transportation network for each road node, as well as their distances from the road node.
These sets are much smaller than the transit-node sets used in TNR. This means that the actual search can be restricted to the smaller public transportation network, skipping the road network by directly “jumping” to the access-nodes of the source and target nodes. The search on the public transportation network thus is a simple multi-modal many-to-many search from the entry-access-nodes of the source to the exit-access-nodes of the target. This approach forces the use of public transportation, so a uni-modal time-independent shortest path search is performed on the road network to check whether the shortest path does not use public transportation at all. For this part, an arbitrary speed-up technique can be chosen, since it is a completely separate search. As access-nodes are precomputed with respect to a certain language, the algorithm accepts only the regular language used during preprocessing and not one given at search time without loss of optimality.

Although ANR has been presented for a continental setting, where the public transportation network is composed of the railway and flight networks and the road network is nation- or continent-wide, we will see later in this work how the approach can be successfully adapted for an urban scenario, where public transportation is more “short-range”, with many short stops.

2.4.4 Core-based Access-Node Routing

Since ANR needs to precompute and store access-nodes for every road node of a potentially continental network, the space consumption of the precomputed data may become intractable.

To mitigate this drawback, ANR has been improved by combining it with Core-Based Routing (thus using contraction on the road network). Precomputation of access nodes is required only for the much smaller core graph obtained after contraction, thus gaining speed-
Figure 9: **Access-Node Routing.** From the source node $s$ the path to its (blue) forward access-nodes is found, as well as from the (red) backward access-nodes of the target to the target. Then the shortest path from the forward access-nodes to the backward access-nodes is computed.

ups both during preprocessing and search time, as well as lowering space consumption. The search algorithm then first determines the relevant core nodes of $s$ and $t$ and then runs a multi-source multi-target ANR search between those, choosing the best complete path from $s$ to $t$. As the paths found must contain a public transportation node, like the original ANR algorithm, the CHASE [29] algorithm is used to quickly get the length of the shortest path using only the road network.
Figure 10: **Core-based Access-Node Routing.** From the source node $s$ the path to its (blue) core-entry-nodes, as well as from the (blue) core-exit-nodes for the target to the target. Then a ANR search is performed, where the road network is actually the contracted core-graph.

### 2.5 Two Way Shortest Path Search

All previously mentioned algorithms and research have been focusing on finding the shortest path in one direction only, from a source $s$ to a target $t$. This is certainly the most natural and obvious part of routing.

On the other hand, there are a number of scenarios where routing only in one direction is not enough. In particular, in the case where parking a private vehicle is involved: when routing from $s$ to $t$ in a multi-modal graph with both private and public transportation modes, a parking location $p$ between $s$ and $t$ might be found for the private vehicle. But since the routing algorithm optimizes mainly for time, this same parking location $p$ might be very inconvenient for the return trip from $t$ to $s$ at a different time of the day (because at least part of the graph
is time-dependent). The need for a more comprehensive, two-way routing approach stems from these kind of scenarios.

To the best of our knowledge, only two approaches have been explored so far. The first is a naive enumerative approach, while the second tackles the problem by using multiple bi-directional searches.

2.5.1 Enumerative approach

Bousquet et al.[30] are (to the best of our knowledge) the first authors to approach the 2-WAY MULTI-MODAL SHORTEST PATH PROBLEM. They present a dynamic label-setting strategy to solve it, requiring at most (3N+3) iterations of the one-way algorithm (an adaptation of Dijkstra’s algorithm), where N is the number of nodes where a transfer between private and public modes can be performed. T is the set of the valid parking locations for the chosen private vehicle (bike or car). The approach consists of 4 steps:

- For the outward trip:
  - The shortest paths tree from the origin O to each parking location \( i \in T \) for the vehicle is computed, storing the corresponding costs (one iteration of the one-way algorithm).
  - The shortest path from each \( i \in T \cup \{O\} \) to the destination D is computed, without using the private vehicle (N iterations of the one-way algorithm, N+1 if \( O \notin T \)).

- For the return trip:
– The shortest paths tree from the destination D to each \( i \in T \cup \{ O \} \) with no vehicle available is computed, using one iteration of the one-way algorithm.

– The shortest paths from each \( i \in T \) to the origin O are computed, considering that a vehicle is available and has to be brought back, requiring N iterations of the one-way algorithm.

They also discuss using a parking spot \( P_0 \) where the vehicle is initially parked and where it has to be left before going back to the origin O. The above steps discuss the case where \( P_0 = O \), so in the other cases the algorithm needs N more iterations of the one-way algorithm to find the shortest path from each \( i \in T \) to O through \( P_0 \). Their approach is thus purely enumerative and not scalable for even medium sizes of the set of transition nodes (52 seconds for a search, with \(|N| = 10\)). Also, they consider only private bikes and not bike sharing, which would need to choose two optimal parking spots for each trip, and they do not take into account other constraints (budget, number of mode changes, etc.).

2.5.2 Regular language constrained approach

Starting from the work of Bousquet et al.[30], Kirchler[2][31] presents a new approach for the 2-WAY MULTI-MODAL SHORTEST PATH PROBLEM (2WMMSPP), providing a more scalable and less time consuming approach and formalizing some of the concepts of the 2WMMSPP.

The proposed basic algorithm to solve the problem divides the two-way trip into four legs, two for the outward path and two for the return one. It then applies an instance of the generalized DIJKSTRA’s algorithm for the RLCSP to each of the four legs at the same time, settling the node with the lowest key among all of the four queues at each iteration, until a node
has been settled by all four. Then the two legs where backward search has been applied need to be computed again with the actual departure times from the parking location, thus getting the actual value of the trip. The four searches continue to find shortest two-way paths until either the queues are empty or the new found solution is worse than the best one so far. Kirchler then also proposes a variation of this algorithm, where instead of RLCSPP-Dijkstra the SDALT algorithm is used for the search. The computational results presented in [2] are very promising, but the approach is very general since it tackles a general problem, the RLCSPPP. We will see how our contribution is more focused and specific in Chapter 4.
CHAPTER 3

PROBLEM DEFINITION

In this chapter, we define the problem to be tackled by stating the underlying assumptions made when developing the approach presented in Chapter 4. We start by giving a formal definition of the problem, which we call the **Effective Time Two-Way Park-aNd-Ride Problem (ET-2WPNRP)**, stating the needed elements and the resulting outcomes we expect when solving it. We then follow up with some clarifications on what we mean by two-way path and on the problem itself. In Section 3.2 we then describe in more detail the scenario we would like to focus on and what features characterize our work, in particular details on which transportation modes can be used and what trip times need to be specified. Finally, we define which kind of parking locations we have decided to take into account when searching for a two-way path and the reasons for this choice.

3.1 **Effective Time Two-Way Park-aNd-Ride Problem**

In this section we want to formally define the problem of finding the two-way path between two locations which minimizes the effective time of the trip, given that the user starts with a private vehicle that needs to be parked in an intermediate location and later retrieved. The key characteristics of this problem are thus the fact that the path to be found is *two-way* and the need for a parking location where to leave the private vehicle. Taking into account these characteristics, the ET-2WPNRP can thus be defined as follows.
Definition 3.1 (Effective Time Two-Way Park-aNd-Ride Problem (ET-2WPNRP)).

Given:

- a directed multi-modal time-dependent graph $G = (V, A)$ consisting of the merged networks of multiple modes of transportation (see Section 1.2.2),
- a set of arc cost functions for the graph, $C = \{c : c : A \to \mathbb{F}\}$,
- a departure (source) node $s \in V$,
- a destination (target) node $t \in V$,
- a latest arrival time at the target node $\tau^t_{maxArr}$ or an earliest departure time from the source $\tau^s_{minDep}$,
- an earliest departure time from the target node $\tau^t_{minDep} > \tau^t_{maxArr}$, or a latest arrival time at the source node $\tau^s_{maxArr}$,
- an initial private vehicle, to be used at the beginning of the outward leg and at the end of the return leg

find a two-way path $\vec{p}$ such that both the outward and the return leg use the same parking node $q = PARKING\_NODE$ (i.e., use two-way Park-aNd-Ride), minimizing its effective time, defined as the sum of the trip duration plus the additional waiting at the source and target nodes.

The four different combinations of latest arrival and earliest departure for $s$ and $t$ are analogous and thus, without loss of generality, we will focus on the scenario with a lastest
arrival time at the target node ($\tau_{\text{maxArr}}$) and an earliest departure time from the target node $\tau_{\text{minDep}}$. In the case where the private vehicle is not used at all (when $q = s$), thus using only public transportation, or used for the whole trip (when $q = t$), the two legs of the two-way trip become independent and can thus be optimized separately using known and fast algorithm for one-way routing. This means that the more interesting case is that where an intermediate parking location is used, as we can easily find the paths using only the private vehicle or only public transportation by running separate searches using known and fast algorithm for one-way routing and then compare the resulting trips to result of our approach. Thus, from now on, we will assume that the used parking location is an intermediate one.

The problem above defined can be seen, to better imagine it, as a two-way combination of a Latest Departure Problem (LDP) (for the outward leg) and an Earliest Arrival Problem (EAP) (for the return leg). The ET-2WPNRP is obviously not just a combination of these two known problems, since the problem lies in optimizing both legs at the same time, as it is not sufficient to find the optimal paths for the two “sub-problems” independently. We are in fact interested in finding not the optimal path, but the optimal two-way path, i.e., a path consisting of an outward leg from the source location to the destination and a return leg from the destination to the source location with optimal effective time.

In a realistic scenario, routing in an urban environment involves multiple modes of transportation: private cars or bicycles, public transportation, bike sharing, walking, etc. Thus we want to focus on multi-modal scenarios, where a user can use different modes of transportation in different parts of his journey. In particular, as we defined in 3.1, we are interested in finding
a two-way path constrained by a parking location. This is the case when the user begins his journey using a private vehicle (e.g., a car, a bike), which has to be parked in some intermediate parking location and later picked up during the return leg. The two-way path will thus actually consist of four sub-paths (see Figure 11), between the source and the parking locations (in both directions) and between the parking and target locations (in both directions).

A very intuitive and easily formalizable definition of a two-way path can be

**Definition 3.2 (Two-way path).** A two-way path $\xrightarrow{\quad} p$ between two nodes, $s$ and $t$, consists of a parking node $v$ and the concatenation of four sub-paths:

- $p_o^1 = (s, ..., v),$
- $p_o^2 = (v, ..., t),$
- $p_r^1 = (t, ..., v),$
- $p_r^2 = (v, ..., s).$

where $p^o = p_o^1 \circ p_o^2$ is the outward leg, while $p^r = p_r^1 \circ p_r^2$ is the return leg.

This means that the parking location adds an important constraint on the return path. It has to be chosen with both legs in mind, and finding the optimal two-way path means finding the parking location that minimizes the objective function. In fact, we can see in Figure 12 how the need to use a parking location ($p_1, p_2, p_3$ in Figure 12) makes optimizing the two legs separately sub-optimal. If we optimize both legs as if they were two separate paths, we could get two different parking locations for the two legs, which is clearly not a viable alternative for the user. When optimizing the outward leg, e.g., the parking location chosen will influence
the return leg, which will have to visit the parking location of the outward path and this may result in a much longer path than needed. On the other hand, if the parking node is the source node (i.e., the private vehicle is never actually used) or the target node (i.e., the vehicle is used until the destination), the problem becomes trivial: it is sufficient to optimize the two legs separately, as if they were two independent trips.

3.2 Main features

The most intuitive features of the problem come from the definition of the ET-2WPNRP itself: the use of a parking location among those available throughout the urban area and the choice of a time window in which to stay at the destination before returning to the departure location. We will here go into more detail on these features to define a specific scenario representing the ET-2WPNRP we are addressing in this work, as well as discussing what kind of transportation modes are of interest and the role a budget might play in this scenario.
Figure 12: **Two-way optimization better than one-way.** We can see how in this example optimizing the path from $s$ to $t$ in both ways leads to a better result than in one-way. The best two-way path uses $p_2$ with a cost of 27, while optimizing the two directions separately leads to using $p_1$ for the outward trip with a cost of 11 and $p_3$ for the return trip with cost 6, which is not admissible since the parking location is not the same. Moreover, optimizing only in one way and adapting the other leg to that choice leads to either using $p_1$ for both legs with a cost of 32 (not optimal), or (when optimizing the return trip) to having no path going through $p_3$ for the outward trip. Thus optimizing in two ways results in an optimal two-way path.

**Modes**

First of all, as an important input, the initial transportation mode has to be provided. This can be either the private car, the private motorcycle or the private bicycle, which has to be parked during the trip. Specifying which type of vehicle to use defines which kind of parking locations are needed and what kind of roads can be used. As well as the starting mode input, we can also make considerations on the later part of the journey using public transportation. In
particular, we should limit the number of mode changes to a sane maximum number and also limit how many times the same mode of transportation can be used in a single leg. For example, a leg where the user has to enter and exit the subway network multiple times is probably not acceptable in terms of user satisfaction. So we limit the available modes of transportation in the following way, for each of the two legs:

- **private car/bike**: \( = 1 \), obviously the private vehicle can only be used at the beginning of the first leg and the end of the second,
- **walking (except simple route transfers)**: \( \leq 2 \),
- **subway/urban railway**: \( \leq 1 \), but changing subway line within the same subway station is not counted towards the number of changes,
- **surface transit**: \( \leq 2 \), buses, trams etc.

**Time**

As defined in 3.1, the search for the optimal two-way path takes two time instants as time constraints for the search: the latest arrival time at the target location and the earliest departure from the target location. This is in fact a very common scenario for the ET-2WPNNRP: if, for example, the user has an appointment downtown, he will want to find the quickest two-way path that allows him to be there on time. He will also know approximately when the appointment will be over, thus the moment from which a return trip needs to be found to go back to the source location. We thus want to minimize the time the user actually “wastes” on the journey, instead of just how much he spends travelling.
Parking locations

A central role in this work is played by parking locations for cars and bicycles. Parking spaces for cars can be of different types: on-street parkings, private parking garages or park-and-ride facilities. The city of Milan, for example, has 21 very big park-and-ride facilities which are usually placed near terminuses. The number of parking spaces of these varies between 320 and 2200, with an average of 851 spaces. In addition to these, there are more than 200 private parking garages, more or less evenly distributed in the metropolitan area and with varying number of spaces available, as well as varying cost per hour (ranging from 1.5 €/h to 5 €/h). We consider these two categories when tackling the ET-2WPNNP, since we are interested in providing routes also using parking locations which are nearer to the city center and not only the few and farther park-and-ride facilities.

If we look at bicycle parking locations (commonly called “bike racks”), we will see that the number of locations is even greater: e.g., Milan has more than 550 bicycle racks distributed throughout the city and the nearby suburbs.

On further inspection, we can also observe that many of these are very near to each other, especially those in proximity of important points of interest or transportation stations (e.g., central station, subway stations with multiple lines, etc.). If we only consider the park-and-ride facilities and leave out parking garages and bicycles, the problem may be solvable by exhaustive enumeration (as presented in [30]), but with mediocre performance and resulting paths. The great amount of candidate parking location creates the need for a purely algorithmic solution, since enumeration does not perform well enough and limits the available routes.
Figure 13: Milan’s private parking garages are spread throughout the city, mainly in the vicinity of important public transit stops, like subway terminuses and stations (map from Google Maps).

Budget

Another interesting aspect of the problem, particularly in European urban settings, is having a budget for the trip. In fact, many cities apply congestion charges or fees to enter specific areas. Moreover, parking the private car in a parking lot with hourly fees can be quite expensive in some central areas, whereas in areas farther from the city center it is cheaper. In the US there are only few examples of congestion charges, but private parking garages are available everywhere as well. Since it is very difficult to get updated fee prices for every parking, one can approximate these price difference by creating several regions within the city with different parking prices. The budget itself can be used either as an additional constraint on the optimal
Figure 14: Milan’s bicycle racks are much more densely distributed in the urban area (map data from Google Maps).

path to be used during the search (pruning paths that violate it while searching), as a post-processing step filtering out the computed alternative optimal paths that are too expensive, as a second criterion for the search algorithm (thus making it a *multicriteria* search algorithm), or even ignored altogether.
Figure 15: Milan’s bicycle racks are not distributed evenly, but rather grouped at important locations (map data from Google Maps).
Figure 16: Milan’s congestion charge area (AreaC) covers the city’s center area and has daily pass prices ranging from 2€ to 5€ (map from OpenStreetMap).
PROPOSED APPROACH

In this chapter we present the approach, and the related algorithm, we developed in order to solve the Effective Time Two-Way Park-and-Ride Problem (ET-2WPNRP) as we defined it in Chapter 3. The main goal of our work is in fact to present and evaluate an exact algorithm to find the shortest paths from a source location to a target location and from the target to the source on mixed transportation networks, optimizing the effective time of the outward and the return leg combined and not the two legs separately.

4.1 Proposed Approach

Our proposed approach to solve the Effective Time Two-Way Park-and-Ride Problem (ET-2WPNRP) builds on and extends the Access-Node Routing (ANR) approach briefly described in Section 2.4.3 to take into account the characteristics of the problem as defined in Section 3.1. In particular, we adapt the access-node computation procedure to cope with a time-dependent road network, which in the original work is assumed time-independent. Moreover, we extend the approach significantly to the case of two-way park-and-ride route planning in an urban scenario, whereas it was originally conceived as a one-way multi-modal approach without any support for park-and-ride and more focused on a rail and flight network combination. We here describe how the approach has been both adapted and extended.
The key idea, inspired by ANR, is the assumption that the road network is used only at the beginning and the end of a journey. This assumption has been limited to the case where the road network is used by private vehicle only at the beginning of the outward leg and the end of the return leg, while it is used solely on foot at the end of the outward leg and the beginning of the return leg. We also adapt the key idea of precomputing “relevant” entry points into the public transportation network, i.e., access-nodes, by adding a second type of access-node which represents a valid parking location. Distances are precomputed to and from each node in the road network, similarly to how Access-Node Routing [1] [28] performs this, but keeping in mind the two-way nature of the problem. The search algorithm can be then divided into different stages, having different segments of the entire networks decoupled by the access-nodes. We thus force the use of a parking location where to leave the private vehicle, since we are not considering parking the vehicle on the roadside or in parking spots found by chance.

We assume to have a multi-modal graph where each valid location in the road network has been connected to the nearest public transportation stops by a transfer arc, as discussed in Section 1.2.2. Nodes in the road network represent intersections and edges represent road segments between those intersections. Each public transportation stop has at least one node that represents being at the stop and there are multiple nodes for the same stop if it has several platforms or lines. These nodes are linked with low-weight edges to the road network. If station entrances or stops are too far away from an existing intersection node, the street arc would be split accordingly to position the entrance or stop more realistically. We thus use a time-dependent (as opposed to time-expanded) model for the graph, similar to the realistic
timetable information model presented in Section 2.4. We can see an example visualization in Figure 17. There are boarding, alighting, dwell, and hop arcs representing public transit trips. These arcs are time-dependent; for example, a boarding arc will search for the next departure time for its associated trip and vary its weight accordingly. The public transportation part of the graph is time-dependent and obviously satisfies the First In First Out (FIFO) property, while, to do so, the road network needs to satisfy the condition on the slope of the arc cost of being $\geq -1$, as we saw in Section 2.4. We will see in Section 5.3 how our approximation method for adding time-dependency to the road network satisfies this condition.

Expanding ANR, we distinguish between two types of access-nodes: access-nodes and Park and Ride (PNR)-nodes. We start by describing how we adapted ANR’s concepts of access-nodes and the procedure to compute them for each road node in a time-dependent road network (opposed to the original time-independent network), before presenting the extensions we have developed to tackle the two-way park-and-ride nature of our problem.

4.1.1 Details on Access-nodes

The access-node set for a road node $v$ can be seen as the set of entry points to the public transportation network which are relevant at least once during the day, for a trip starting in $v$. A node $w$ is relevant for a road node $v$ if there is at least one shortest path starting in $v$ during the day that uses $w$ as its entry point for the public transportation network.
Figure 17: Example of a graph of a street network, linked to the graph of a transit network graph. The black part of the network represents the road network (dashed arcs are walk-only), the purple nodes represent transit stops together with their departure and arrival nodes (in blue). The remaining red and green arcs and nodes represent transit lines, dashed lines representing staying on the bus/tram.

**Access-Node candidates**

Before defining which nodes are access-nodes, we first need to define which nodes can potentially be access-nodes. These nodes are called *access-node candidates* [1] and are defined as the public transit stops which are connected to the road network.
Definition 4.1 (Access-Node candidates). Let $G = (V,A)$ be a multi-modal graph. A node $v \in V$ is called access-node candidate if the following properties hold:

1. The node $v$ is not part of the road network ($v \neq \text{ROAD\_NODE}$)

2. The node $v$ is linked to the road network, i.e., there is a neighbor $w$ of $v$ for which $w = \text{ROAD\_NODE}$

The set of access-node candidates is denoted by $A$.

Precomputing and storing the distances to all of these nodes from every road node would need a very large amount of space and time, considering also that not every candidate will actually be used to perform shortest path searches during the day. Thus, from the sets of access-node candidates, we need to derive the sets of actual access-nodes, which will be used by the search algorithm.

Access-Nodes

Once we know which is the set of potential access-nodes, we can define which nodes actually are access-nodes. These can be defined[1] as follows.

Definition 4.2 (Access-Node). Let $G = (V,A)$ be a multi-modal graph with an access-node candidate set $A$ and let $v \in V, v = \text{ROAD\_NODE}$ be a node belonging to the road network (thus also the parking nodes). Then a node $a \in A$ is called access-node for $v$, if there exists another access-node candidate $b \in A$ and a departure time $\tau$ for which the shortest path from $v$ to $b$ at time $\tau$ uses the node $a$ to enter the public transportation network.

The set of access-nodes for a road node $v$ is denoted by $A(v)$. 
Backward Access-Nodes [1]

When considering trips ending in \(v\), the set of relevant access-nodes is obviously not the same as the set of relevant nodes for trips starting in \(v\), since a public transportation stop could be used in a shortest path in one way, but not in the other. We thus need to compute the set of exit points of the public transportation network which are relevant at least once during the day, for every road node. These are called backward access-nodes. The algorithm used for the computation of the backward access-nodes is the same as for the forward access-nodes and simply performs a forward access-nodes computation on the backward graph \(\overleftarrow{G}\).

**Definition 4.3 (Backward Access-Node).** Let \(G = (V, A)\) be a multi-modal graph with an access-node candidate set \(A\) and let \(v \in V, v = ROAD\_NO\) be a node belonging to the road network.

Then a node \(a \in A\) is called backward access-node for \(v\), if \(a\) is a forward access-node for \(v\) in the backward graph \(\overleftarrow{G}\).

The set of backward access-nodes for a road node \(v\) is denoted by \(\overleftarrow{A}(v)\).

4.1.2 Extension to Park-aNd-Ride: PNR-Nodes

In the one-way original ANR approach with a time-independent road network, access-nodes are enough to skip the road network and restrict the search to a many-to-many search from \(A(s)\) to \(\overleftarrow{A}(t)\). In our two-way park-and-ride setting, we need both the concept of access-nodes and a new concept of PNR-nodes, i.e., park-and-ride locations which are relevant for a two-way shortest path search at least once during the day. Summarizing, PNR-nodes are used as parking locations for two-way trips, backward access-nodes for the target are used as exit points
from the public transportation network during the outward leg and forward access-nodes for the target are used as entry points during the return leg. Moreover, once the user has parked the private vehicle, he has to enter the public transportation network and he will use it also to get back to the chosen parking location. Thus, access-nodes are also necessary for each of the PNR-nodes: forward access-nodes for PNR-nodes are used for the outward leg, and backward access-nodes for the return leg.

**PNR-Nodes candidates**

Similarly to access-node candidates, PNR-node candidates are the parking locations connected to the road network. These parking locations can be either for cars (PNR-nodes) or for bicycles, i.e., “bike racks” (Bike-PNR-nodes, or *BPNR-nodes*). The definition is the same for both types of parking, so we will simply use the “PNR” to refer to both, if not otherwise specified.

**Definition 4.4 (PNR-NODE CANDIDATES).** Let $G = (V, A)$ be a multi-modal graph. A node $p \in V$ is called PNR-node candidate if the following properties hold:

1. The node $p$ is not part of the road network ($p \neq ROAD\_NODE$)
2. The node $p$ is linked to a parking location, i.e., there is a neighbor $q$ of $p$ for which $q = PARKING\_NODE$

The set of access-node candidates is denoted by $P$. 

PNR-Nodes

Once we have established the set of PNR-node candidates, we need to define PNR-nodes, i.e., PNR-node candidates which are relevant for a two-way shortest path search at least once during the day.

**Definition 4.5 (PNR-NODE).** Let $G = (V, A)$ be a multi-modal graph with an PNR-node candidate set $P$ and let $v \in V$, $v = \text{ROAD\_NODE}$ be a node belonging to the road network.

Then a node $p \in P$ is called PNR-node for $v$, if there exists another PNR-node candidate $q \in Q$ and a departure time $\tau$ for which the shortest two-way path from $v$ to $q$ at time $\tau$ uses the node $p$ as its parking location.

The set of PNR-nodes for a road node $v$ is denoted by $P(v)$.

Since we consider the two-way path for the computation of PNR-nodes, there is no distinction among forward and backward PNR-nodes like there is for access-nodes. This seems clear when one thinks of the role the two types of nodes have in the two-way path: the entry- and exit-nodes for and from the public transportation network for the target node can be different (one can take the subway for the first leg and a bus on the return leg), but the parking node has to be the same, since the vehicle has to be picked up at the same location it has been left before.

4.2 Preprocessing

The preprocessing stage is the most importante stage. In this phase the sets of access-nodes and PNR-nodes for every road node are computed, which will then be used during the actual
optimization phase. Thus, this phase will have a great influence on the results of the search. We start by describing how access-nodes

\subsection{Known methods for computing access-nodes}

Pajor \cite{1,28} presents two approaches to compute the sets of forward and backward access-nodes $A(v)$ and $\overrightarrow{A}(v)$ (as defined in Section 4.1.1) for all $v \in V$ with $v = ROAD\_NODE$: an exact approach and an approximate approach. Both approaches involve computing profile searches at some point during the procedure, the difference being when and how.

**Exact computation of access-nodes**

This approach applies the definition of access-nodes, thus computing a one-to-all time-dependent multi-modal profile search from every road node $v$, computing all needed access-nodes during the search. This is performed with a multi-label-correcting approach. The biggest problem with this approach appears when having a fairly high number of road nodes, since for each of them a profile search needs to be performed. In the case of Milan, \textit{e.g.}, the number of road nodes is in the order of the hundreds of thousands, and bigger cities have significantly more.

**Approximated computation of access-nodes**

The approximate approach has a somewhat opposite point of view. Instead of computing the set of access-nodes for every road node $v$ $A(v)$, the idea is to compute for every access-node candidate $a \in A$ the inverse relation $A^{-1}(a)$, \textit{i.e.}, the set of road nodes for which $a$ is an access-node. $A^{-1}$ is defined as follows:
∀v ∈ V, ∀a ∈ A : v ∈ A^{-1}(a) ⇐⇒ a ∈ A(v) \quad (4.1)

Once $A^{-1}(a)$ has been computed, we can invert it to obtain $A(v)$.

Figure 18: Approximating Access-Nodes. Since the upper bounds on both paths from $v$ to $a_1$ which use $a_2$ and $a_3$ are longer than the direct path from $v$ to $a_1$, node $v$ is contained in $A^{-1}(a)$, and $a_1$ would thus be added to the access-node set of $v$. In reality, $a_1$ is not an access-node for $v$, since at no time of the day the direct path is shorter than the indirect paths through $a_2$ and $a_3$. Thus $a_1$ is redundant.

The approach is not exact only because $A^{-1}(a)$ might contain more nodes than necessary. This is not a problem for the correctness of the procedure, since it does not skip needed access-nodes and just may find redundant ones.

The procedure works in two steps. First, given an access-node candidate $a ∈ A$, a full multi-modal backward profile search on the public transportation subnetwork of $\overleftarrow{G}$ is performed,
starting from \( a \). This gives us travel time functions \( f_b \) for each access-node candidate \( b \in A \), representing the time needed to get from node \( b \) to node \( a \) in the subnetwork for any time of the day.

The algorithm looks for all road nodes \( v \in V \) for which \( a \in A(v) \), i.e., road nodes for which another access-node \( b \in A \) is reached by entering the public transportation subnetwork in \( a \) at least once during the day, which can be found in \( A^{-1}(a) \). The second step of the procedure is a uni-modal time-independent many-to-all Dijkstra search, restricted to the road subnetwork. The road network in [1] is assumed time-independent, thus simplifying the preprocessing phase considerably. The priority queue of the search is initialized with the other access-node candidates \( b \in A \), having keys equal to the upper bound of their time profile computed in step 1, \( \overline{f}_b \). The given access-node \( a \) is inserted as well, but with a key equal to 0 and a flag \( \text{covered} = \text{true} \), which will be propagated through the backward search on \( \overrightarrow{G} \).

Every time a road node \( v \in V \) is settled, it is inserted in the set \( A^{-1}(a) \) if and only if its flag \( \text{covered}(v) = \text{true} \), i.e., if it pays off using the road network to get to node \( a \) instead of entering the public transportation network from another node and using the subnetwork to get to \( a \). Viceversa, for all nodes \( v \in V \) with \( \text{covered}(v) = \text{false} \), it always pays off to use another access-node to enter the public transportation network instead of \( a \), because the shortest path from \( v \) to \( a \) uses some other access-node candidates at any time during the day. This is true because by using the upper bound of the path duration from some \( b \in A \) to \( a \), the worst connection during the day is considered.
The procedure turns out to be not exact, since an access-node candidate $a$ does not have to be an actual access-node for every $v \in V$ with the covered flag set. In fact, using upper bounds instead of the exact profile functions might be too conservative.

Performance with respect to the exact approach improves drastically, since profile searches are performed only for access-node candidates and on the smaller public transportation network. Nonetheless, performance still degrades badly when the size of the public transportation network increases and the profile searches become the bottleneck of the preprocessing phase.

4.2.2 Determining PNR-Nodes

When determining the set of PNR-nodes, we can take some ideas from the access-node procedures. The key differences with respect to the computation of access-nodes, are the type of network on which the procedure is performed and the type of profile search performed. In fact, computation of the PNR-nodes is performed on a time-dependent road network, opposed to a time-independent road network for access-nodes, thus increasing complexity of the procedure significantly. Moreover, the profile searches do not search for an upper-bound on a one-way path, but rather an upper-bound on a two-way path.

As in the approximate computation approach for access-nodes, instead of computing the set of PNR-nodes for every road node $v \ P(v)$, the idea is to compute for every PNR-node candidate $p \in P$ the inverse relation $P^{-1}(p)$, i.e., the set of road nodes for which $p$ is a relevant PNR node. $P^{-1}$ is defined as follows:

$$\forall v \in V, \forall p \in P : \ v \in P^{-1}(p) \iff p \in P(v)$$ (4.2)
Once $P^{-1}(p)$ has been computed, we can invert it to obtain $P(v)$. The procedure works in two steps. First, given an PNR-node candidate $p \in P$, a full multi-modal backward profile search on the public transportation subnetwork of $\overleftarrow{G}$ is performed, starting from $p$. This gives us travel time function $f_q(p)$ for each PNR-node candidate $q \in P$, representing the time needed to get from node $q$ to node $p$ in the subnetwork, for any time of the day.

The algorithm looks for all road nodes $v \in V$ for which $p \in P(v)$, i.e., road nodes for which another PNR-node $q \in P$ is reached by entering the public transportation subnetwork in $p$ at least once during the day, which can be found in $P^{-1}(p)$.

The second step of the procedure can be separated into two cases: either we are precomputing PNR-nodes for a time-independent or a time-dependent road network.

**Bicycle PNR-nodes**

In the case of the private vehicle being a bicycle for example, we will model the road network as time-independent. Thus, the second step consists in a uni-modal time-independent many-to-all Dijkstra search, restricted to the road subnetwork, for each $p \in P$.

The priority queue of the search is initialized with the other PNR-node candidates $q \in P$, having keys equal to the sum of upper bounds of the time profiles of the two-way path from $p$ to $q$, computed in step 1, $\overleftarrow{f}_{p,q} = \overleftarrow{f}_p(q) + \overleftarrow{f}_q(p)$. The given access-node $p$ is inserted as well, but with a key equal to 0 and a flag $covered = true$, which will be propagated through the backward search on $\overleftarrow{G}$.

Every time a road node $v \in V$ is settled, it is inserted in the set $P^{-1}(p)$ if and only if its flag $covered(v) == true$, i.e., if it pays off using the road network to get to node $p$ from $v$ instead of
entering the public transportation network from another PNR node and using the subnetwork to get to $p$. Vice versa, for all nodes $v \in V$ with $\text{covered}(v) == false$, it always pays off to use another PNR node to enter the public transportation network instead of $p$, because the shortest path from $v$ to $p$ uses some other PNR-node candidates at any time during the day. This is true because by using the sum of the upper bounds of the path duration from some $q \in P$ to $p$ and the of the reverse path, the worst combination of connections during the day is considered. Thus, this procedure as well turns out to be not exact, since a PNR-node candidate $p$ does not have to be an actual PNR node for every $v \in V$ with the $\text{covered}$ flag set, since not only are we using upper bounds, but a sum of two upper bounds.

**Car PNR-nodes**

When the private vehicle used is, e.g., a car, we want to model the road network as time-dependent, since this view of urban roads is more realistic. Because of this, we cannot use the same approach as before to compute the set of relevant PNR-nodes, but need to adapt it to time-dependency. In fact, applying a many-to-all Dijkstra as with time-independent networks would result in wrong PNR-nodes. This is due to two facts: the search only considers the path from the PNR-node candidate to the source and not the inverse path (which could increase the travel time to and from the PNR-node candidate significantly!), and the search is time-dependent, thus the same path can have very different travel times during the day.

The first solution that comes to mind is performing two time-dependent profile-searches for every PNR-node candidate, one on the forward graph and one on the backward graph. These lead to time travel functions to (from) each node from (to) the PNR-node candidate, of which
we are interested in keeping upper \((f_{s,p})\) and lower bounds \((f_{s,p})\). Then, for each road node \(v\), a PNR-node candidate \(p \in P\) is a PNR node for \(v\) if for each other PNR-node candidate \(q\) we have that \(f_{p,q} + f_{s,p} \leq f_{s,q}\), i.e., there would be some times during the day where the two-way path from \(s\) to \(q\) using \(p\) as PNR node is more convenient than using the road network to get to \(q\) directly.

An alternative would be to use a time-dependent many-to-all label-correcting algorithm on the forward graph starting from the set \(P\), where for every node a label representing the cost to get from a PNR-node candidate \(p\) to that node is stored, and one on the backward graph, where the labels added represent the cost to get from the node to a PNR-node candidate \(p\). The arc costs are the lower bounds of the travel time functions for the arc.

Then for each node we can evaluate if there is a time \(\tau\) during the day for which a given PNR-node candidate \(p\) is used in the shortest path from the node to some other PNR-node candidate \(q\). The problem here is that the comparison for every time of the day becomes quickly very expensive.

### 4.3 Search for an optimal two-way path

Once access-nodes and PNR-nodes have been computed in the preprocessing phase described above, we need to actually search for shortest paths using user input. We will thus assume to be given a multi-modal graph \(G = (V, E)\) and sets of forward and backward access-nodes \((A(v)\) and \(\overrightarrow{A}(v))\), as well as sets of PNR-nodes \(P(v)\) for each road node \(v \in V, v = ROAD\_NODE\). The user input consists in a source node \(s\), a target node \(t\), a maximum arrival time at the target \(\tau_a\) and a minimum departure time from the target \(\tau_d\).
Since we want to treat the road network of the private vehicle as time-dependent instead of time-independent, we can not store distances to the access and PNR-nodes, as shown in [1], since these distances change during the day. We thus store only the nodes and have to compute the distance during the search search.

The two-way path the algorithm is looking for, from source $s$ to target $t$ using a parking location $p$, can be divided into eight parts:

1. from the source $s$ to one of its PNR-nodes $p \in P(s)$,
2. from the PNR-node $p \in P(s)$ to one of its forward access-nodes $a_p \in A(p)$,
3. from the access-node $a_p$ of the PNR-node $p$ to one of the backward access-nodes of the target, $b_t \in \overleftarrow{A}(t)$,
4. from the backward access-node of the target $b_t \in \overleftarrow{A}(t)$ to the target $t$,
5. from the target $t$ to one of its forward access-nodes $a_t \in A(t)$,
6. from the target’s access-nodes $a_t \in A(t)$ to one of the backward access-nodes of the source’s PNR-node $b_p \in \overleftarrow{A}(p)$,
7. from the backward access-node of the PNR-node $b_p \in \overleftarrow{A}(p)$ to the source’s PNR-node $p \in P(s)$,
8. from the PNR-node $p \in P(s)$ to the source $s$

A visual example of the segments of a two-way path is shown in Figure 19.

A very simple approach to find the two-way path consists in having multiple multi-modal time-dependent Dijkstra’s algorithm instances running independently for the two legs of the
Figure 19: **Segments of a two-way path.** We can observe the various segments of a two-way path we defined. The (black) outward leg consists of a segment from the source to one of its PNR-nodes (the one minimizing the two-way effective time) (1), a segment from the PNR-node to one of its forward access-nodes (2), a segment from the access-node to one of the target’s backward access-nodes (3) and finally from the backward access-node to the target itself (4). Analogously, the (red) return leg consists of a segment from the target to one of its forward access-nodes (5), a segment from the access-node to one of the backward access-nodes of the PNR-node (6), a segment from the backward access-node to the PNR-node (7) and lastly a segment from the PNR-node to the source node (8).
PNR-nodes, which in turn feeds the results to a search from the forward access-nodes of the PNR-nodes to the actual PNR-nodes $P(s)$. These distances can then finally be used to initialize a multi-label Dijkstra search with $|P(s)|$ labels from the PNR-nodes of the source to the source itself, thus getting the cost from the source to the target of the best path for each of the PNR-nodes.

Similarly, we can perform an analogous search from the target $t$ to the source $s$ for the return path, resulting in travel times from $t$ to $s$ for each PNR node used, but first getting to the target’s forward access-nodes, then to the backward access-nodes of the source’s PNR-nodes, to the PNR-nodes and finally to $s$. We can then simply sum up the forward and backward travel times for each PNR node and choose the PNR node that minimizes this combined time.

This approach obviously is not efficient, as it has to compute travel times for every PNR node and compare it against the others. If the number of combinations is low, this can be done quickly, but the approach does not scale for a high number of PNR-nodes. Therefore we need a more efficient approach.

Finding the minimum two-way travel time during the actual search, and not during a post-processing phase, is thus a key goal we want to achieve.

**Simultaneous two-way algorithm**

Starting from the intuitive algorithm explained above, a more efficient approach has been developed, optimizing both directions of the trip simultaneously. This approach allows to find the minimum two-way path during the actual search, instead of comparing alternatives during post-processing and choosing the best one (basically enumerating the options). Our proposed
two-way algorithm, denoted by Two-Way Search Algorithm (2WSA) works as follows (with reference to the pseudo-code in Algorithm 3). First of all, two searches are applied simultaneously on the graph, one for each of the two legs of the trip (outward and return). We call $D_o$ and $D_r$ the searches for the outward and return legs of the two-way path, respectively.

Because of how we defined our problem, specifically because of the time constraints we defined in Chapter 3, the search for the outward path will look for a path arriving by a certain time $\tau_a$ while the search for the return path will look for a path departing after a certain time $\tau_d$. Thus, the search instance $D_o$ will actually not be a forward search, but will be performed on the backward graph $\overrightarrow{G}$ starting from the node $t$ at time $\tau_a$. The search for the return path $D_r$ on the other hand is a forward search from $t$ starting at time $\tau_d$.

The searches are performed through multi-modal labelling algorithms working similarly to multi-modal Dijkstra’s algorithm ([24]). The difference is the ability to produce multiple alternative paths from the source to the destination. In fact, instead of having one label per node as Dijkstra’s algorithm does, the two searches keep sets of labels for each node. These sets are all initialized to empty sets, except the source’s one which is set to a single 0. The priority queue also contains the additional information together with the nodes to be explored. We use these labels to mark the costs of the paths for each of the PNR-nodes the algorithm finds a path for (in one direction), thus each node will have at most $|P|$ labels for each of the two searches, where $P$ is the set of PNR-nodes of the source node. Before reaching a PNR-node (i.e., when still in the public transportation and walking segments of the trip), only one label is used and thus the searches behave like multi-modal Dijkstra searches.
The two searches use a conceptually shared priority queue, which in reality is composed of two priority queues, one for each search. We execute these two searches on the graph at the same time: at each iteration of our algorithm, we choose the search algorithm among $D_o$ and $D_r$ which has the node $v$ that is going to be settled next in its queue (i.e., the node with the lowest key among the two at the top of the two priority queues). The chosen search executes one step, settling node $v$ and relaxing all its outward arcs (i.e. adding the heads of those arcs to the priority queue).

After reaching all PNR-nodes, each search continues on with the use of the private vehicle. This part of the search uses multiple labels for each node. In particular, for each PNR-node that can be used to get to a node $v$, a label is kept for node $v$ representing the cost of the path from the target $t$ to the node $v$ by using that PNR-node. This allows us to find a path for each PNR-node and direction, one by one in increasing order of cost until the stopping condition is met.

After a number of steps, the source node $s$ will be settled by $D_o$ or by $D_r$, using a certain PNR node $p$ during the path in the corresponding direction (thus storing the cost label using $p$). A classic Dijkstra's algorithm would stop at this point, when the source $s$ is settled, and the cost of the path found would be the optimal solution. But since we want the minimum two-way path, our algorithm will continue the search and each time the source $s$ is settled by one of the two searches, the PNR-node $p$ used for the corresponding path is stored (only the optimal path for each PNR-node and direction is stored, as we are not interested in having
more). When $D_o$ settles $s$ with a certain PNR node $p$ and $D_r$ has already settled $s$ with $p$ (or vice versa), a new candidate two-way path using PNR node $p$ as a parking node has been found.

To evaluate the travel time of the found candidate two-way path, the travel times of the two legs are summed. If the total travel time of the found solution is lower than the cost $\mu$ of the best two-way path found up to this point (or it is the first one), then the found solution is stored as the new best two-way path and $\mu$ is updated with its cost. If on the other hand the search that got to the source location arrived with a cost that, summed with the minimum cost of all the one-way paths for the other direction found so far, is worse than the best solution found so far, the searches stop. It is in fact guaranteed that all future solutions will be worse than that sum and thus also worse than the stored solution, since both searches will only increase the costs of future explored nodes. The minimum cost of all one-way paths for the other direction is in fact a lower bound on the actual cost of that path.

Assuming $P_o$ and $P_r$ are the two sets of PNR-nodes for which the searches $D_o$ and $D_r$ have found a path to the source, $w^o_p$ and $w^r_p$ are the costs of the one-way paths found in the two directions that use PNR-node $p \in P_o$ and $p \in P_r$ respectively, let $p^* \in P_o \cap P_r$ be the PNR-node used by the current best solution found. Supposing that the return search just arrived at the source node $s$ using a new path with parking node $q \in P_r \setminus P_r$, then the stopping condition of the algorithm can be expressed as:

$$w^r_q + \min_{p \in P_o} w^o_p \geq w^o_{p^*} + w^r_{p^*}$$  \hspace{1cm} (4.3)
Analogously, if the outward search just arrived at the source node \( s \) through \( q \in P_o \setminus P_r \), then the stopping condition of the algorithm can be expressed as:

\[
\omega_q + \min_{p \in P_r} \omega^r_p \geq \omega^r_{p^*} + \omega^o_{p^*}
\]  
(4.4)

Also, if one the priority queues is empty, or if the cost of currently explored node is greater than the best solution minus the minimum cost of the paths of the other search, the corresponding search stops and only the other continues. When both searches stop, the stored optimal solution is the shortest two-way path.

Figure 20: **Example instance graph.** This graph will be used in the 5 steps shown below.
Example instance

We now briefly describe how the stopping condition works by using an example graph (shown in Figure 20). The graph is greatly simplified, as we are interested in showing the stopping condition and how two-way paths are found rather than how the algorithm traverses arcs. The graph in Figure 20 is explored by the two search instances $D_o$ (red) and $D_r$ (blue) until one of the two ($D_r$) finds a path to the source. The corresponding parking node $p_2$ is stored in $D_r$’s set of used PNR-nodes together with the cost of the path found, and the minimum cost of $D_r$ is set to the path’s cost ($\min_{D_r} = 8$, Figure 21). $D_o$ and $D_r$ continue exploring until $D_r$ finds a new path to the source, but this time using $p_1$ with cost 9, thus adding $p_1$ to its set of used PNR-nodes but not decreasing the minimum cost of $D_r$ (Figure 22). After more exploring, $D_o$ finds a path to the target which uses $p_1$ and has cost 10, thus adding it to $D_o$’s set and

![Diagram](image)

(a) Backward graph (outward trip)  
(b) Forward graph (return trip)

Figure 21: Step 1. $D_r$ finds a path from $t$ to $s$ using $p_2$ with cost 8.
Figure 22: **Step 2.** $D_r$ finds a path from $t$ to $s$ using $p_1$ with cost 9.

setting the minimum cost to 10 (Figure 23). Since $p_1$ is already contained in $D_r$’s set of used parking nodes, we have found a new two-way path candidate with cost $10 + 9 = 19$ using $p_1$ as its parking location. Next, $D_o$ finds another path to the target node (Figure 24) using $p_2$,

Figure 23: **Step 3.** $D_o$ finds a path from $s$ to $t$ using $p_1$ with cost 10. The two-way path using $p_1$ becomes the current best solution with cost 19.
which is added to $D_o$’s set (not decreasing the minimum cost). Since $p_2$ is already in $D_r$’s set of used parking locations, we have found a new two-way path candidate using $p_2$ which has a cost of $10 + 8 = 18$ and thus becomes the new best candidate found so far. Finally, $D_o$ finds a

Figure 24: **Step 4.** $D_o$ finds a path from $s$ to $t$ using $p_2$ with cost 10. The two-way path using $p_2$ becomes the current best solution with cost 18.

path using $p_3$ as its parking node with cost 12 and adds $p_3$ to its set of used parking locations (Figure 25). $p_3$ is not in $D_r$’s set, but the sum of the cost of the path using $p_3$ (12) plus the minimum cost of all one-way paths found so far (in this case, it’s the path found by $D_r$ using $p_2$, with cost 8) is greater than the cost of the best two-way path found so far (18). Thus, the stopping condition is fulfilled and we can stop the search. In fact, $D_o$ and $D_r$ can only find paths to the source node which have a cost $\geq 12$ from this point on, thus it is not possible to find a two-way path having a cost lower than 20. This means we have found the optimal two-way path from $s$ to $t$, having cost 18 and using $p_2$ as the parking location.
(a) Backward graph (outward trip)  
(b) Forward graph (return trip)

Figure 25: **Step 5.** $D_o$ finds a path from $s$ to $t$ using $p_3$ with cost 12.

**Complexity analysis**

To analyze the complexity of our proposed two-way algorithm, we will analyze only one of the two search directions (the return search), as the total complexity will be double the complexity of each of the two searches. The return search can be divided into two main parts: before and after reaching the PNR-nodes. Up until the search reaches the PNR-nodes, it is analogous to a multi-modal time-dependent Dijkstra’s algorithm instance with a complexity of $O(|A| \log |V|)$ where $|A|$ is the number of arcs and $|V|$ the number of vertices of the entire graph. On a closer look, though, we can see how the search until the PNR-nodes, by clearing and reinitializing the priority queue for the first three segments, is better described by three smaller multi-modal time-dependent Dijkstra’s algorithm instances: the first one on the foot network (from $t$ to its access-nodes) with a complexity of $O(|A_f| \log |V_f|)$ where $|A_f|$ and $|V_f|$ are the number of arcs and vertices of the foot subnetwork only, the second one on the public transportation network (from $s$’s access-nodes, seen as one single node, to the relevant PNR-
nodes’ backward access-nodes) with a complexity of \(O(|A_t| \log |V_t|)\) where \(|A_t|\) and \(|V_t|\) are the number of arcs and vertices of the public transportation subnetwork only and the third again on the foot network (from the backward access-nodes, seen as one single node, to the relevant PNR-nodes) with a complexity of \(O(|A_f| \log |V_f|)\). Thus the complexity of the first part of the search is \(O(|A_f| \log |V_f| + |A_t| \log |V_t|)\).

The second part of the search, from the source’s PNR-nodes to the source itself, is performed with a search which is analogous to a multi-label multi-modal time-dependent Dijkstra’s algorithm, which has a complexity of \(O(d|A| \log |V|)\) where \(d\) is the number of labels. In our case, the number of labels will be at most equal to the number of PNR-nodes, \(|P|\), and thus the complexity of the second part of the search is \(O(|P||A_p| \log |V_p|)\), where \(|A_p|\) and \(|V_p|\) are the number of arcs and vertices of the private vehicle’s subnetwork only. Consequently, the total complexity of each of the two searches is \(O(|P||A_p| \log |V_p| + |A_f| \log |V_f| + |A_t| \log |V_t|)\) and the total worst-case complexity of the proposed search algorithm is \(O(|P||A_p| \log |V_p| + |A_f| \log |V_f| + |A_t| \log |V_t|)\), which is better than the worst-case complexity of the approach proposed in [30] \((O(|L||A_p| \log(|V_p|) + |L||A_f| \log(|V_f| + |A_t| \log |V_t|)))\), where \(|L|\) is the number of all parking locations in the graph) and in [2] \((O(m \log(n) + n m \log(n)))\), where \(m\) and \(n\) are the arcs and nodes of the product graph). Since the complexity is dominated by the term \(O(|P||A_p| \log |V_p|)\), a very important goal of our approach is to keep \(|P|\) low. Lower sizes of the set of PNR-nodes can be achieved by having a preprocessing phase, which determines \(P\), that discards all useless PNR-node candidates. Although not explicitly present in the complexity analysis (since they do not affect the worst case), also (backward) access-nodes play an important role in reducing
computation time, as they need to be settled before proceeding to the next phase and thus the fewer there are, the sooner the next phase can start. Consequently, also keeping the sizes of those sets to the minimum is a key factor in performance of the approach.
Algorithm 3 Two-Way Park and Ride Routing

1: \(Q_o\) ← priority queue
2: \(Q_r\) ← priority queue
3: \(M_o\) ← set
4: \(M_r\) ← set
5: \(F_o\) ← map
6: \(F_r\) ← map
7: \(spt\) ← MultiLabelShortestPathTree
8: \(Q_o\).insert(target, 0)
9: \(Q_r\).insert(target, 0)
10: \(status = WALKING\)
11:
12: while not \(Q_o\).isEmpty() and \(Q_r\).isEmpty() do
13:   outward ← \(Q_o\).minimumKey() < \(Q_r\).minimumKey()
14:   if outward then
15:     \(Q\) ← \(Q_o\)
16:     \(M\) ← \(M_o\)
17:   else
18:     \(Q\) ← \(Q_r\)
19:     \(M\) ← \(M_r\)
20: end if
21: state ← \(Q\).dequeue()
22: if \(spt\).dominated(state) then
23:   continue;
24: end if
25: if checkMilestone(state, status, \(M\), \(Q\)) then
26:   \(Q\).clear()
27:   \(Q\).addAll(\(M\))
28: else
29:   if outward then
30:     \(arcs\) ← incoming arcs \((v,w)\)
31:   else
32:     \(arcs\) ← outgoing arcs \((v,w)\)
33: end if
34: for all \((v,w)\) in \(arcs\) do
35:   if not \(Q\).contains(w) then
36:     \(Q\).insert(w, dist(s,v) + \(c_{vw}\))
37:     \(pre(w)\) ← \(v\)
38:   else
39:     if dist(s,v)+\(c_{vw}\) < dist(s,w) then
40:       \(Q\).decreaseKey(w, dist(s,v)+\(c_{vw}\))
41:       \(pre(w)\) ← \(v\)
42: end if
43: end if
44: end for
45: \(spt\).add(state)
46: end if
Algorithm 3 Two-Way Park and Ride Routing (continues)

47: \textbf{if} state.vertex == source and status == DRIVING and exists(state.pnrNode) \textbf{then}
48: \hspace{1em} \textbf{if} outward \textbf{then}
49: \hspace{2em} \text{F}_o.put(state.pnrNode, state)
50: \hspace{2em} \textbf{if} \text{F}_r.contains(state.pnrNode) \textbf{then}
51: \hspace{3em} newTime \leftarrow state.travelTime + \text{F}_r.getMinimumTravelTime()
52: \hspace{3em} \textbf{if} newTime \geq bestTime \textbf{then}
53: \hspace{4em} \text{stop}()
54: \hspace{3em} \textbf{else}
55: \hspace{4em} bestTime \leftarrow newTime
56: \hspace{2em} \textbf{end if}
57: \hspace{1em} \textbf{end if}
58: \hspace{1em} \textbf{else}
59: \hspace{2em} \text{F}_r.put(state.pnrNode, state)
60: \hspace{2em} \textbf{if} \text{F}_o.contains(state.pnrNode) \textbf{then}
61: \hspace{3em} newTime \leftarrow state.travelTime + \text{F}_o.getMinimumTravelTime()
62: \hspace{3em} \textbf{if} newTime \geq bestTime \textbf{then}
63: \hspace{4em} \text{stop}()
64: \hspace{3em} \textbf{else}
65: \hspace{4em} bestTime \leftarrow newTime
66: \hspace{2em} \textbf{end if}
67: \hspace{1em} \textbf{end if}
68: \hspace{1em} \textbf{end if}
69: \hspace{1em} \textbf{end if}
70: \textbf{end while}
Algorithm 4 checkMilestone function

1: if $Q == Q_o$ then
2:   if status == WALKING and $M_o.containsAll(target.backwardAccessNodes)$ then
3:     status = TRANSIT
4:     return true
5:   else
6:     if status == TRANSIT and $M_o.containsAll(source.pnrNodes.accessNodes)$ then
7:       status = TRANSFER_TO_PNR
8:       return true
9:     else
10:    if status == TRANSFER_TO_PNR and $M_o.containsAll(source.pnrNodes)$ then
11:       status = DRIVING
12:       return true
13:     end if
14:   end if
15: else
16:   if status == WALKING and $M_r.containsAll(target.accessNodes)$ then
17:     status = TRANSIT
18:     return true
19:   else
20:     if status == TRANSIT and $M_r.containsAll(source.pnrNodes.backwardAccessNodes)$ then
21:       status = TRANSFER_TO_PNR
22:       return true
23:     else
24:       if status == TRANSFER_TO_PNR and $M_r.containsAll(source.pnrNodes)$ then
25:         status = DRIVING
26:         return true
27:     end if
28:   end if
29: end if
30: end if
31: return false
CHAPTER 5

IMPLEMENTATION AND EXPERIMENTAL RESULTS

In this chapter, we describe in detail how the preprocessing procedures and the search algorithm presented in Chapter 4 have been implemented and applied on real-world data, as well as show the results produced by the application of the presented approaches to the data. We will start in Section 5.1 by introducing OpenTripPlanner (OTP), a multi-modal trip planning platform we extended to use our proposed approach presented in Chapter 4. We then continue in Section 5.3 by explaining the input data for the experiments, as well as what the setup of the experiments consisted of. Lastly, in Section 5.5, we show figures of the application of the algorithm to the real-world data described in Section 5.3, comparing it with some alternatives.

5.1 OpenTripPlanner

The work presented so far has been implemented as an addition to a well-known Open Source multi-modal trip planning platform called OpenTripPlanner (OTP) [32], which was launched in 2009 and at first supported only by Portland Oregon’s transport agency, TriMet. The project is now deployed for a number of cities around the globe, both officially by transit agencies and unofficially by companies or organizations, thanks to the use of open standards. The motivation for the choice of OTP is twofold. First of all, testing our solution within a real-world routing platform (opposed to an ad-hoc, optimized solution) adds value to the viability of the solution itself, since it is not only viable for “clean” settings. Secondly, since
solving the Effective Time Two-Way Park-aNd-Ride Problem (ET-2WPNRP) gives end-users many advantages and makes their life easier, the work done on the implementation of our approach can be the starting point for more future improvements and for a future inclusion in the project itself, with many positive effects for end-users.

OpenTripPlanner’s backend is developed in Java and provides a REpresentational State Transfer (REST) Application Programming Interface (API) service for trip planning, as well as a Javascript client with visual maps (much like more “commercial” options, e.g., Google Maps [33]). OpenTripPlanner imports data from open standard file formats, like General Transit Feed Specification (GTFS) [34] for transit data and OpenStreetMap [35] for map data.

Currently, OpenTripPlanner uses a single time-dependent graph containing both the time-independent street and time-dependent transit networks. Walk-only and bicycle-only trips are planned using the A* algorithm [36] with a Euclidean heuristic or contraction hierarchies [37].

5.2 Implementation

As described in Chapter 4, our proposed approach consists of two phases: a preprocessing phase and a real-time optimization phase. Both phases thus have been implemented extending OTP as needed, but at the same time keeping it compatible with current developments of the platform, in order to make it easier to keep it updated and/or merging it with the original project. In Appendix A we give a more detailed description on how to use the source code of our implementation and where to find it, while here we concentrate more on the key features and characteristics of the implementation. Since the platform (OTP) is developed in Java on the server and routing side, all of our implementation has been developed in Java as well.
5.2.1 Graph preprocessing

Preprocessing in OpenTripPlanner is performed by the GraphBuilder class, which in turn uses so-called graph builder modules to perform various tasks. Modules can be easily created by extending the GraphBuilderModule class, and then added to the preprocessing phase by simply adding it to the GraphBuilder’s list of modules to execute. For our approach, we added four new modules to the preprocessing phase:

- **AccessNodeModule**, implements the computation of the access-nodes sets for each of the road nodes, of the car parking locations and the bicycle parking locations, as described in Section 4.2.1.

- **PNRNodeModule**, implements the computation of the car PNR-nodes sets for each of the road nodes, as described in Section 4.2.2.

- **BikePNRNodeModule**, implements the computation of the Bike-PNR-nodes sets for each of the road nodes, as described in Section 4.2.2.

- **StreetDirectionModule**, computes the direction of each road arc with respect to the city center, setting the direction flag for each of them.

The first three modules, computing access- and PNR-nodes, would need to use a profile routing algorithm for the computation of the upper bounds used during the preprocessing phase to initialize the priority queues of the second step of the procedure. Unfortunately, OTP does not provide such an algorithm for use during preprocessing yet and the implementation of such an algorithm would be out of scope for this work. Thus, we implemented our preprocessing
phase using a classic Dijkstra’s algorithm. In order to obtain upper bounds on the shortest paths, this Dijkstra search only uses the upper bound of each arc’s cost (i.e., the upper bound of each time-dependent arc’s cost, since for time-independent ones it will actually be equal to the cost itself). Thus, the resulting upper bounds are obviously higher than if an actual profile search would be used and the sizes of the resulting access- and PNR-node sets are greater. Consequently, this will also affect the real-time optimization phase, since it has to explore more nodes before proceeding to the next step. In conclusion, the results of our implementation here presented, using OTP, will suffer from this drawback. Nevertheless, we will show later in this chapter how our approach is viable even with this issue, thus making the case that the algorithm as presented in Section 4.3 would have even better real-time performance than those here shown.

5.2.2 Two-way search algorithm implementation

Once the input data has been processed and the graph builder has created a graph object file, the bidirectional search algorithm presented in Section 4.3 to be used to search for the shortest two-way path on the graph object described in Section 5.2.1. An Algorithm interface has been added to OTP in order to generalize the search algorithm to be used by the application server, which can be selected among the original modified A* and the proposed implementation (called PNRDijkstra) using a command line switch (--twoWayRouting).

The proposed implementation uses the two-way algorithm presented in Section 4.3 and thus keeps track of the two searches (for the outward and return legs) using two priority queues implemented as binary heaps. It also keeps track of the current goal nodes in so-called milestone
sets, one for each leg. Both searches start from the target location on foot (since it is the last part of the outward leg and the first part of the return leg), the outward search being flagged as arrive-by (i.e., performed on the backward graph). Each time the outward search finds a backward access-node of the target, it adds it to its milestone set; the same is done by the return search, except that its goal nodes are the forward access-nodes of the target instead of the backward ones. When a search fills its milestone set with all goal nodes (in this case, the backward, or forward, access-nodes of the target) or when its priority queue is empty and there are no new nodes to be explored, the milestone set flushes its nodes into the priority queue with their cost serving as key in the queue. The search then changes its status, switching to public transportation as mode of transportation and the new goal nodes are the forward access-nodes of the PNR-nodes of the source (for the outward leg search) or their backward access-nodes (for the return leg search). Again, the milestone sets are filled up and flushed into the priority queue, then the status changes, the new goal nodes are the PNR-nodes of the source and the mode of transportation is walking again, but this time as a mode change between public transportation and the private vehicle. The process repeats itself one more time, when the transportation mode changes to the private vehicle (car or bicycle) and the goal node is just the source node.

When one of the two searches reaches the source location, the PNR-node used by the path that reached the source node is stored in a set of used parking locations (each of the two searches uses a different set). When also the second search arrives at the source node, it adds the PNR-node its first path used to the set of used parkings and then checks whether the parking location is also present in the set of parking nodes of the first search. If it is, a new two-way path has
been found. The cost of this new path is computed by summing the costs of the two legs and if the cost is lower than that of the currently best solution, the best solution is updated. The searches stop as soon as one of the gets to the source location with a cost that, summed with the minimum cost of all the legs found so far, is worse (greater) than the best solution found so far, as explained in Section 4.3’s paragraph on our proposed two-way algorithm.

5.3 Input Data

As inputs for our experiments we used various sources of data for the city of Milan, Italy. Map data for an area of about a third of the city has been exported from the official OpenStreetMap website [38]. Transit data has been obtained from Open Data provided by the transit agency of the city, Azienda Trasporti Milanesi (ATM) [39]. The transit data has been retrieved in the GTFS format [34], which specifies a common standard for how public transit schedules and their associated geographic information are stored, in order to make it easier for developers to consume transit data published by public transit agencies. From these types of datasets, OTP produces a graph object after a preprocessing phase (briefly explained in Section 5.2.1), which is then stored to file. The graph file produced during the preprocessing stage can then be passed to an application server instance, which provides the REST API service and calls the routing procedures when the requests arrive.

Estimating road arc slow-down factors for different time bands

In our approach we consider the road network to be time-dependent for cars, contrarily to how most of the previous works handle it, thus we need to actually have time-dependency in the car network part of the graph used, i.e., the street arcs need to have different costs at different
times of the day. Since retrieving real-time data to use during the real-time optimization phase has not been possible, due to obvious reasons of non-availability, we decided to approximate traffic data with a more empirical approach. We decided to use an approach which has been used in practice by urban mobility researches as an approximation when more detailed data is not available and which adds time-dependency to the network by using street direction as an indicator of which streets need to be delayed in various time bands. The approach consists in using a coefficient for traffic delay on road arcs traversed by car which is dependent on both time of the day and direction of the arc. Our approximated model, in fact, makes the simple assumption that roads that go towards the city’s center are more heavily affected by traffic during the morning rush hours, while roads that “exit” the city suffer more traffic during the evening rush hours. Moreover, roads that approximate a circle around the city’s center suffer during both rush hours. Thus, to decide what coefficient to use to multiply the cost of the road arc, we not only need to know the different time windows representing rush hours, but also the direction of the road: entering, exiting or circular.

During preprocessing a flag is added to each street arc representing the direction of the street. The computation of this flag relies on simple geometry: if $c$ is the city’s center and $(u, v)$ is the road arc, we simply have to look at the geographic distances of the head and tail of the arc from the center. If $v$ is farther (geographically) from $c$ than $u$, the road is exiting, and vice versa the road is entering. Obviously, a circular road will not be a perfect arc of a circle, but rather it will approximate one. Thus, we say the road arc is circular if $d_u - \varepsilon \leq d_v \leq d_u + \varepsilon$, where $d_u = \text{dist}(u, c)$ is the geographic distance from $u$ to $c$ and $d_v = \text{dist}(v, c)$ the geographic
distance from \( v \) to \( c \). Updating the other two definitions, we say that a road arc is *entering* if \( d_v < d_u - \varepsilon \) and *exiting* if \( d_v > d_u + \varepsilon \) (Figure 26). Obviously, this computation needs the geographic coordinates of the city center, in order to do these simple comparisons. These are
passed as build parameters for the graph builder using the `build.json` file it uses to load the parameters.

**Street network delay**

The street directions computed are then used to delay the road network during rush hours. In particular, after carefully examining the city of Milan’s documents on urban traffic planning (available at [40]), the road network has been delayed during the real-time optimization phase in the following time bands:

- **Morning rush hour**: between 7:30am and 9:30am we reduce the speed on road segments entering the city and on road segments that are circular around the city center by 40%. This reduction is not abrupt, but rather performed gradually, by linearly reducing the maximum speed of the road segment from 100% to 60% of its original value in the span of the first 30 minutes (i.e., from 7:30am to 8:00am). The maximum speed is then increased again linearly from 60% to 100% of its original value during the last 30 minutes of the time window (i.e., from 9:00am to 9:30am);

- **Evening rush hour**: lasting from 4:30pm to 7:30pm, with speed reduced by 40% in the same way as for the morning rush hour time window (linearly reducing and increasing the maximum speed during the first and last 30 minutes of time window, respectively).

The time-dependent arc costs resulting from this approximation method satisfy the First In First Out (FIFO) property, as the slopes are always $\geq -1$ (only when linearly decreasing the arc cost the slope is $-1$, which is still acceptable to satisfy the property).
5.4 Computational experiments

The goal of our experiments has been to prove that our approach is viable in a real-world scenario. To achieve this, we have set up a number of experiments, described in Section 5.4.1, which use transportation data freely available to everyone using open standards (Section 5.3). All of our experiments were conducted on a basic laptop computer featuring a Dual-Core Intel® Core™ 2 Duo P8700 processor at 2.53GHz, with 3MiB level 2 cache and 4 GiB of main memory. While OTP itself has support for multithreading in its server implementation, for our experiments we used a single-threaded automated testing procedure, in order to bypass the use of OTP’s APIs and directly use the routing procedures. No ad-hoc optimizations have been made to the implementation other than those described in Chapter 4.

For each of the scenarios described in Section 5.4.1 we performed 100 experiments, generated randomly by a suitable procedure, the results of which are presented in Section 5.5 together with figures of the preprocessing phase. For each of these 100 experiments, a two-way path has been computed using the implementation of our proposed approach, as well as the implementation of the basic version of the algorithm, proposed in Section 4.3. Moreover, upper bounds on the optimal solutions have been computed and have been compared against the optimal solutions found by the two-way optimization algorithm. We are able to show figures only of our proposed approach and not of a detailed comparison with previous works from literature, as the authors of [31] and [2] could not provide us their implementation due to confidentiality reasons.
5.4.1 Scenarios

To test our proposed approach, we first generated a number of test instances which varied the following input properties:

- departure (source) location \( s \),
- arrival (target) location \( t \),
- latest arrival time at the target location \( \tau_{\text{maxArr}}^t \),
- earliest departure from the target location \( \tau_{\text{minDep}}^t \),
- initial private vehicle (car or bicycle).

The values of the input properties have been randomly chosen for every instance using different ranges depending on the scenario:

- **Commute**: this scenario is a realistic simulation of the use case the ET-2WPNRP represents best and which is most interesting for our work; the latest arrival time at the target location is randomly chosen in the interval at \([8:00\, am, 9:30\, am]\) and the earliest departure from the target location is randomly chosen in \([5:30\, pm, 7:30\, pm]\); source locations are randomly chosen from areas of the suburbs, while target locations are chosen randomly in downtown areas, thus simulating a normal commute to a workplace. In this scenario we will thus focus on the rush hour traffic’s influence on the two-way trip, since many road arcs entering the city will be delayed.
• **Commute with fixed stay**: this scenario is a particular case of the *Commute* scenario, where the earliest departure from the destination is always set eight and a half hours after the latest arrival at the destination.

• **Night Out**: this scenario is not as realistic as the aforementioned *Commute* scenario, but still representative of certain interesting use cases, in particular a night out in the city center for a dinner or an event. The latest arrival time at the target is set randomly during the evening ([7:00 pm, 9:00 pm]) and the earliest departure from the target during the night ([11:00 pm, 1:00 am]), choosing the locations in the same way as for the *Commute* scenario. Thus, in this scenario we can see how the availability of public transportation for both legs of the two-way trip is essential and one-way route planning is not able to choose the parking location accordingly.

• **Night Out with fixed stay**: this scenario is a particular case of the Night Out scenario, where the earliest departure from the destination is always set three and a half hours after the latest arrival at the destination.

All of these have been combined with both alternatives for the initial private vehicle: 50 of each scenario use the private car and 50 use the private bicycle. In Figure 27 we can see the suburb areas used to generate the source locations and the city center area used to generate the target locations.

### 5.5 Computational results

In this section we present the results of our experiments, in terms of computational time spent for the preprocessing phase as well as average computation times for the optimization
Figure 27: **Source and target location areas for experiments.** The suburb areas used to generate source locations are marked in green, while the city center (target locations) is marked in red. The outmost border shows the limits of the map used for the experiments (map data from OpenStreetMap).

phase. We also analyze the resulting two-way paths, looking at the average effective times of each scenario and the composition of the paths, in terms of how much time is spent on each transportation mode.

### 5.5.1 Preprocessing

The preprocessing phase has a great influence on the actual search, and thus we examine the output of preprocessing in order to better contextualize the search results we present later.
Preprocessing has been performed starting from the data described in Section 5.3 and the resulting graph presented statistics described in Table I.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertices</td>
<td>73,247</td>
</tr>
<tr>
<td>Arcs</td>
<td>174,461</td>
</tr>
<tr>
<td>Access-node candidates</td>
<td>2,024</td>
</tr>
<tr>
<td>Access-nodes per road vertex (PRV)</td>
<td>1,232</td>
</tr>
<tr>
<td>Backward access-nodes PRV</td>
<td>348</td>
</tr>
<tr>
<td>PNR-node candidates</td>
<td>99</td>
</tr>
<tr>
<td>PNR-nodes PRV</td>
<td>94</td>
</tr>
<tr>
<td>Bike-PNR-node candidates</td>
<td>310</td>
</tr>
<tr>
<td>Bike-PNR-nodes PRV</td>
<td>154</td>
</tr>
<tr>
<td>Preprocessing time (in min)</td>
<td>106</td>
</tr>
</tbody>
</table>

We can see how the lack of a profile routing functionality for the preprocessing phase influences the access-node and PNR-node sets: on average almost all candidate car parkings are used as PNR-nodes and almost half of bicycle parkings are used as Bike-PNR nodes. Also 60% of the access-node candidates are used as access-nodes. The number of access-nodes and (Bike-)PNR-nodes directly affects performance, as these nodes need to be settled before proceeding to the next segment of the search. Thus, the greater the number of nodes in these sets
is, the longer it will take for each segment. The first conclusion is thus that the profile routing functionality, which is missing in our implementation, can be a more than valid improvement to the preprocessing phase and consequently to search. In fact, in our implementation we used multi-modal time-dependent Dijkstra’s algorithm instances applied on the graph having arc costs equal to the upper bound on the original graph’s arc cost to compute upper bounds for the distances among couples of transit stops or parking locations. The tighter these upper bounds are, the more candidate nodes for the PNR-node and (backward) access-node sets are discarded and thus the sets will be smaller. Thus, profile routing would decrease these sets dramatically, as the upper bounds found by using it are exact upper bounds: the maximum cost of the optimal path between two nodes during a specific time window (in our case, a day).

In addition to the “clean” preprocessing presented in Table I, we also generated graphs with artificially decreased upper bounds during the preprocessing phase. In fact, since profile routing is not available to be used during preprocessing, the upper bounds computed are a sum of upper bounds on the single arcs and are thus a very high upper bound on the complete path from one node to another. Lowering the upper bound would thus decrease the sizes of the sets to be computed. We artificially decreased the upper bounds by means of a coefficient the computed upper bound is multiplied by, which has been set to six different values (figures in Table II). The first three values are fixed for all upper bounds on the paths (0.5, 0.25 and 0.125), while the other three (Var-1, Var-2 and Var-3) are variable and depend on the length of the upper bound. In particular, the value of the coefficient for Var-1 is interpolated, where the minimum upper bound is kept untouched and the maximum is divided by 10. Var-2 behaves similarly,
but the minimum upper bound is divided by 2 and the maximum by then (thus the others have coefficients which are interpolated between 0.5 and 0.1). Lastly, \( \text{Var-3} \) interpolates between a coefficient of 1 and a coefficient of 0.25, but the coefficient is then squared before multiplying the upper bound by it. From Table II we can see how, even with such rudimentary attempts, the decreased upper bounds lead to a decrease of the set sizes. This shows again how the use of profile routing, which provides exact upper bounds for the preprocessing phase, would decrease the numbers dramatically and consequently improve performance of the actual search.

<table>
<thead>
<tr>
<th>Coefficient</th>
<th>0.5</th>
<th>0.25</th>
<th>0.125</th>
<th>Var-1</th>
<th>Var-2</th>
<th>Var-3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access-nodes per road vertex (PRV)</td>
<td>1,053</td>
<td>945</td>
<td>1,022</td>
<td>1,046</td>
<td>1,053</td>
<td>1,011</td>
</tr>
<tr>
<td>Backward access-nodes PRV</td>
<td>352</td>
<td>254</td>
<td>345</td>
<td>334</td>
<td>348</td>
<td>356</td>
</tr>
<tr>
<td>PNR-nodes PRV</td>
<td>92</td>
<td>90</td>
<td>92</td>
<td>92</td>
<td>92</td>
<td>92</td>
</tr>
<tr>
<td>Bike-PNR-nodes PRV</td>
<td>31</td>
<td>2</td>
<td>1</td>
<td>295</td>
<td>297</td>
<td>295</td>
</tr>
<tr>
<td>Preprocessing time (in min)</td>
<td>108</td>
<td>100</td>
<td>96</td>
<td>120</td>
<td>109</td>
<td>106</td>
</tr>
</tbody>
</table>

### 5.5.2 Two-way path two-way algorithm

We start by examining the performance of our proposed implementation of the algorithm solving the Effective Time Two-Way Park-and-Ride Problem (ET-2WPNRP) (as presented in Chapter 4 and implemented as described in Section 5.2). Table III reports compu-
tation time statistics for the optimal two-way paths in the *Commute* scenarios, while Table IV reports the same statistics for the *Night Out* scenarios. We can immediately observe how performance is consistent among all scenarios, as performance does not depend on the scenario itself. We can also observe how the initial mode of transportation chosen has a great impact on performance: choosing the car results in computation times that are on average less than a third of those obtained when choosing a bicycle instead. This is a very intuitive result, since the Bike-PNR-node sets are greater than the PNR-node sets and the bicycle road network is not delayed by rush hour traffic. Both facts imply that the algorithm explores more of the bicycle road network before switching to other modes of transportation, thus increasing computation times.

**TABLE III: COMPUTATION TIME, COMMUTE SCENARIO**

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Commute</th>
<th>Commute (fixed stay)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Car</td>
<td>Bicycle</td>
</tr>
<tr>
<td>Initial mode</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average (in s)</td>
<td>7.6</td>
<td>19.6</td>
</tr>
<tr>
<td>Max (in s)</td>
<td>10.5</td>
<td>26.4</td>
</tr>
<tr>
<td>Min (in s)</td>
<td>5.7</td>
<td>14.4</td>
</tr>
<tr>
<td>50th percentile</td>
<td>7.6</td>
<td>19.2</td>
</tr>
<tr>
<td>75th percentile</td>
<td>8.3</td>
<td>21.9</td>
</tr>
<tr>
<td>90th percentile</td>
<td>8.6</td>
<td>25.2</td>
</tr>
</tbody>
</table>
Another interesting observation can be made observing that the 90th percentiles of the computation times are very close to the average, especially for cars, thus showing one more time how consistent the performance of the algorithm is regardless of the input data. More detailed illustrations of the distributions of computation times can be seen in Figure 28 and Figure 29.

Performance is obviously not on par with one-way routing, but given the added complexity of computing a two-way trip with a parking location constraint, we can definitely say that the presented computational results are very promising. In fact, improving the computation time can be achieved in various ways, especially during the preprocessing phase to decrease the size of the access-node, PNR-node and Bike-PNR-node sets. Moreover, in a more real-world application, OTP would be running on a more performing server, with more computing power.

### TABLE IV: COMPUTATION TIME, NIGHT OUT SCENARIO

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Night Out</th>
<th>Night Out (fixed stay)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial mode</td>
<td>Car</td>
<td>Bicycle</td>
</tr>
<tr>
<td><strong>Average (in s)</strong></td>
<td>7.1</td>
<td>24.1</td>
</tr>
<tr>
<td><strong>Maximum (in s)</strong></td>
<td>12.8</td>
<td>34.3</td>
</tr>
<tr>
<td><strong>Minimum (in s)</strong></td>
<td>5.4</td>
<td>17.2</td>
</tr>
<tr>
<td><strong>50th percentile (in s)</strong></td>
<td>7.0</td>
<td>23.3</td>
</tr>
<tr>
<td><strong>75th percentile (in s)</strong></td>
<td>7.7</td>
<td>27.8</td>
</tr>
<tr>
<td><strong>90th percentile (in s)</strong></td>
<td>8.6</td>
<td>30.1</td>
</tr>
</tbody>
</table>
Figure 28: **Commute scenario computation times distribution.** Shows the count of how many test examples out of 50 for cars and 50 for bikes are in each range of computation time (in milliseconds). The algorithm’s performances are consistent both for cars and bicycles, although for bicycles the distribution is a bit more spread out with respect to cars.

and memory, thus running faster in general. Since optimizing for two ways instead of one (which naturally increases complexity) is a very recent research direction, these results can be seen as a promising start towards future improvements.

After showing performance figures for the computation of optimal two-way paths, we now focus on the outputs of these computations, *i.e.* the two-way paths. To do so, we gathered the effective times of the resulting paths, divided into outward and return legs. We can see statistics for the results of the two-way and one-way optimizations for the *Commute* scenarios in Table V, while the results for the *Night Out* scenarios are presented in Table VI.
To give a better idea of the optimality of the resulting paths, we compared the proposed approach to an upper bound on the optimal solution. This upper bound is given by the minimum of optimizing the outward leg, using the found parking location for the return leg, and optimizing the return leg, using the found parking location for the outward leg. In Table VII we can see the average durations of the trips found using the proposed approach compared to the average of the above described upper bounds, averaged across all experiments. Moreover, we also compared trips found and computation times to the basic version of our approach presented in Section 4.3, where the searches in the two directions are performed sequentially.
TABLE V: TRIP DURATION, COMMUTE SCENARIOS

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Commute</th>
<th></th>
<th>Commute with fixed stay</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Leg</td>
<td>Outward</td>
<td>Return</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Outward</td>
<td>Return</td>
</tr>
<tr>
<td>Car</td>
<td>Average</td>
<td>0:42:23</td>
<td>1:01:24</td>
</tr>
<tr>
<td></td>
<td>Maximum</td>
<td>1:08:34</td>
<td>1:42:18</td>
</tr>
<tr>
<td></td>
<td>Minimum</td>
<td>0:29:05</td>
<td>0:34:47</td>
</tr>
<tr>
<td></td>
<td>Median</td>
<td>0:42:04</td>
<td>1:00:50</td>
</tr>
<tr>
<td>Bicycle</td>
<td>Average</td>
<td>0:40:39</td>
<td>0:42:36</td>
</tr>
<tr>
<td></td>
<td>Maximum</td>
<td>0:57:17</td>
<td>0:58:49</td>
</tr>
<tr>
<td></td>
<td>Minimum</td>
<td>0:24:55</td>
<td>0:28:09</td>
</tr>
<tr>
<td></td>
<td>Median</td>
<td>0:40:11</td>
<td>0:41:58</td>
</tr>
</tbody>
</table>

We can see how the upper bounds are on average around 22% worse than the optimal solutions. Thus, our proposed approach can have a significant impact with respect to simply optimizing one way and using the same parking location for the other direction. By looking at the distributions in Figure 30 and Figure 31, we can see how the proposed approach outputs optimal paths which are in the majority of the cases sensibly better than the upper bound, especially when using the car as the first vehicle.
TABLE VI: TRIP DURATION, *NIGHT OUT* SCENARIOS

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Night Out</th>
<th>Night Out with fixed stay</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Outward</td>
<td>Return</td>
</tr>
<tr>
<td><strong>Leg</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Car</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average</td>
<td>0:45:16</td>
<td>0:49:03</td>
</tr>
<tr>
<td>Maximum</td>
<td>1:06:24</td>
<td>1:11:26</td>
</tr>
<tr>
<td>Minimum</td>
<td>0:29:21</td>
<td>0:35:27</td>
</tr>
<tr>
<td>Median</td>
<td>0:44:53</td>
<td>0:46:07</td>
</tr>
<tr>
<td><strong>Bicycle</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average</td>
<td>0:39:08</td>
<td>0:46:53</td>
</tr>
<tr>
<td>Maximum</td>
<td>0:58:10</td>
<td>1:16:23</td>
</tr>
<tr>
<td>Minimum</td>
<td>0:29:43</td>
<td>0:34:09</td>
</tr>
</tbody>
</table>

TABLE VII: COMPARISON OF PROPOSED TWO-WAY ALGORITHM WITH RESPECT TO UPPER BOUND.

<table>
<thead>
<tr>
<th></th>
<th>Car</th>
<th>Bicycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Two-way trips (average)</td>
<td>1:00:49</td>
<td>0:57:51</td>
</tr>
<tr>
<td>Basic two-way algorithm, computation time (average, in s)</td>
<td>15.0</td>
<td>36.1</td>
</tr>
<tr>
<td>Simultaneous two-way algorithm, computation time (average, in s)</td>
<td>7.39</td>
<td>22.3</td>
</tr>
<tr>
<td>Upper bound, trips (average)</td>
<td>1:15:10</td>
<td>1:09:35</td>
</tr>
<tr>
<td>Upper bound, computation time (average, in s)</td>
<td>0.61</td>
<td>0.65</td>
</tr>
<tr>
<td>Upper bound vs Two-way trips (average)</td>
<td>123.6%</td>
<td>120.3%</td>
</tr>
</tbody>
</table>
Figure 30: *Commute* scenario upper bound duration difference distribution. Shows the count of how many test examples out of 50 for cars and 50 for bikes are in each range of trip duration difference (in seconds) from the upper bound.

Figure 31: *Night Out* scenario upper bound duration difference distribution. Shows the count of how many test examples out of 50 for cars and 50 for bikes are in each range of trip duration difference (in seconds) from the upper bound.
Figure 32: CDF of dequeued states, Basic vs Simultaneous algorithm, by car. Shows the distribution of the dequeued states comparing the Basic and the Simultaneous algorithm when the initial mode used is the car. The two blue data series represent the simultaneous version of the algorithm in the Commute and Night Out scenarios, while two red ones represent the basic version.

Figure 33: CDF of dequeued states, Basic vs Simultaneous algorithm, by bicycle. Shows the distribution of the dequeued states comparing the Basic and the Simultaneous algorithm when the initial mode used is the bicycle. The two blue data series represent the simultaneous version of the algorithm in the Commute and Night Out scenarios, while two red ones represent the basic version.
<table>
<thead>
<tr>
<th>Algorithm, scenario</th>
<th>Runtime (in s)</th>
<th>Dequeued states</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Car</td>
<td>Bicycle</td>
</tr>
<tr>
<td>Basic, Commute scenario</td>
<td>13.3</td>
<td>24.6</td>
</tr>
<tr>
<td>Simultaneous, Commute scenario</td>
<td>7.1</td>
<td>19.6</td>
</tr>
<tr>
<td>Basic, Night Out scenario</td>
<td>16.7</td>
<td>37.7</td>
</tr>
<tr>
<td>Simultaneous, Night Out scenario</td>
<td>6.7</td>
<td>20.3</td>
</tr>
</tbody>
</table>

In Figure 32 and Figure 33 we can see distributions of the dequeued states during the execution of the basic algorithm described in Section 4.3 and the simultaneous algorithm, divided by initial mode of transportation: the car in Figure 32 and the bicycle in Figure 33. We can immediately see how the two blue data series, representing executions of the simultaneous algorithm, in both cases dequeue less states than the basic algorithm (in two shades of red). This can be also seen in Table VIII: on average, the simultaneous algorithm explores about 33.5% less states than the basic algorithm. Thus we can see how the search for a two-way path with the simultaneous exploration of both directions and the use of a stopping condition to terminate the search as soon as possible has a sensible impact on the number of states to be processed.

In addition to just the duration of the trips, we also observed the composition of the two-way trips in terms of how much time is spent on each transportation mode. We can see the
average composition for the *Commute* scenario in Figure 34 and the composition for the *Night Out* scenario in Figure 35.

![Car Composition](image1)

![Bicycle Composition](image2)

**Figure 34: Two-way trip composition for the *Commute* scenario.** Figure 34a represents the composition when the private vehicle used is a car, while Figure 34b represents the case the bicycle as the private vehicle. The composition shown is of both legs of the two-way path computed by 2WSA combined and represents how much of the effective time is spent on each of the transportation modes listed. “Other” indicates mode changes and waiting times.

Overall, we can see how the resulting paths’ effective times are very similar among the two scenarios. There is a greater difference when comparing the initial vehicles used: the private car is used for a slightly longer period of time than the private bicycle. In the case of the *Commute* scenario, this can be easily explained by the fact that the car road network is delayed, while for bicycles it is not. On the other hand, in the case of the *Night Out* scenario, the car is faster than the bicycle, so the car will be parked nearer to the destination while the bicycle will be parked nearer to the departure location. From the visualizations of the two-way path compositions we
can also see how an average of about 13% of the effective time is spent on waiting (either at public transportation stops or at the source and target locations) or changing modes, about 9% when commuting and 16% when visiting the city at night. This is an intuitive result, as public transportation is more available during the day than the evening and night. On an average of about 100 minutes for the entire two-way path, this translates into 13 minutes of waiting and mode changing time. Thus about 7 minutes per leg are spent on mode changes and waiting (on average), showing how the solutions found by our proposed algorithm tend to have only short “dead” times in the two-way path.

We now take two example cases of a commute scenario, routing with the input data shown in Table IX.
TABLE IX: EXAMPLES FOR THE *COMMUTE* SCENARIO, BY CAR

<table>
<thead>
<tr>
<th>ID</th>
<th>Source</th>
<th>Target</th>
<th>Arrive before</th>
<th>Depart after</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>45.510657, 9.230731</td>
<td>45.469907, 9.192945</td>
<td>9:09 AM</td>
<td>5:39 PM</td>
</tr>
<tr>
<td>C2</td>
<td>45.483607, 9.232085</td>
<td>45.478827, 9.192367</td>
<td>9:19 AM</td>
<td>5:49 PM</td>
</tr>
</tbody>
</table>

The resulting two-way path of C1 can be seen in Figure 36, while the two one-way optimized paths for C1 computed by the one-way version of our algorithm can be seen in Figure 37. Analogously, the resulting two-way path of C2 can be seen in Figure 38, while the two one-way optimized paths for C2 computed by the one-way version of our algorithm can be seen in Figure 39.

We can clearly see the advantage of the two-way trip, which uses the same parking location for both legs while the two one-way paths use different parking locations, thus making their solutions inadmissible for the ET-2WPNRP. Moreover, the effective times of the single legs of the two-way path are only slightly worse than their one-way counterparts, thus our algorithm finds a two-way path with only minor setbacks with respect to effective time. This is an important achievement, since the constraint of using the same parking location makes the problem much more complex.
Figure 36: **Example of an optimal two-way trip for C1 obtained by using our proposed two-way algorithm (2WSA), i.e. optimizing both legs together.** The blue star indicates the parking location.

Figure 37: **Example of two one-way optimized legs for the two-way trip C1.** The one-way version of our algorithm (1WPR) is used to compute the optimal legs of a two-way trip separately, thus resulting in two different parking locations being used. The blue star indicates the parking location.
Figure 38: **Example of an optimal two-way trip for C2 obtained by using our proposed two-way algorithm (2WSA), i.e. optimizing both legs together.** The blue star indicates the parking location.

Figure 39: **Example of two one-way optimized legs for the two-way trip C2.** The one-way version of our algorithm (1WPR) is used to compute the optimal legs of a two-way trip separately, thus resulting in two different parking locations being used. The blue star indicates the parking location.
CHAPTER 6

CONCLUDING REMARKS

In this thesis we presented a specific routing problem, called Effective Time Two-Way Park-and-Ride Problem (ET-2WPNR), which has important real-world applications. This problem accounts for the situation in which a commuter has to plan a two-way trip and wants to be in the city center for a specific amount of time, and thus wants to arrive before a certain time $\tau_a$ and wants to depart from there not earlier than a later time $\tau_d$, to get back to the source location. In this common case, the user would obviously prefer arriving as close as possible to the maximum arrival time, while departing as late as possible. Analogously, during the return trip he would prefer arriving as soon as possible, while departing not earlier than $\tau_d$. Since traffic plays a big role in mobility in metropolitan areas, Park and Ride is becoming more and more popular amongst commuters. Using an intermediate parking location on the outward trip sets an important constraint on the route to be taken on the return trip, since the private vehicle needs to be retrieved during the return leg. In Chapter 3 we defined the problem in detail, describing its main characteristics and specifying the focus of this work. One of the most important aspects investigated is the presence of a time-dependent road network, which increases the complexity of the problem even further. Thus, the solution approach presented addressed the ET-2WPNR not only considering a network with fixed schedules, as the public transportation network is, but also a time-dependent network.
To solve this common problem, in Chapter 4 we thus proposed a two-phased approach, using a preprocessing phase to compute important nodes for each of the road nodes and an optimization phase using these nodes to find the best two-way trip between two locations. While the preprocessing phase is performed once for all before the actual query, the optimization phase is performed at query time. We described in detail how the preprocessing phase uses upper bounds on paths between so-called candidate nodes to find those that are actually relevant at least once during the day. These are then stored for each road node. We then described our proposed algorithm to compute a two-way trip consisting of two legs (outward and return), each of which has a segment traversed by private vehicle (car or bike), a segment traversed by public transportation and a segment traversed on foot. The proposed algorithm simultaneously computes the two legs, finding the combination of them that minimizes the overall effective time of the two-way trip. The algorithm works by simultaneously searching for both legs of the two-way trip, using reversed arcs for the outward leg (as it is an arrive-by search). When both searches reach the source node, a new candidate two-way path has been found, but the two searches continue until the candidate two-way path is worse than the lower bound on future candidates. Our approach has been implemented in Java extending the open source journey planning platform OpenTripPlanner (OTP), in order to show how the approach is promising in a real-world application scenario. The presented implementation has then been tested on a common laptop by generating a number of test examples on a graph representing a third of the city of Milan.
In Chapter 5 we compared the proposed approach to the basic algorithm described in Section 4.3, obtaining a 33% reduction of the explored states on average. Moreover, we also compared the solutions found by the proposed algorithm to upper bounds on the optimal solutions, given by the minimum of optimizing the outward leg and using the found parking location for the return leg and the opposite optimization. The comparison in Chapter 5 shows how the optimal solutions found by our proposed approach are about 20% better than the upper bounds. We thus obtain a confirmation of the need to optimize the two-way trip and of the feasibility of the approach we presented. Our approach is in fact very promising for solving the ET-2WPNRP, especially when the initial vehicle used is a car, where the PNR-node sets are smaller on average than the Bike-PNR-node sets and thus the algorithm terminates in only a few seconds on average. By analyzing the composition of the two-way paths, we could also observe how the bicycle is used for a longer percentage of time during the trips compared to the car. We can also see how the paths use public transportation for a greater part of the trip for the Commute scenario, in particular when using a car as the private vehicle. Moreover, the waiting and mode changing time for the Night Out scenario is almost double as much as for the Commute scenario, due to the lower frequency of public transportation.

During the development of the presented approach, as well as the associated implementation, a number of possible future improvements have been identified. First of all, it was not possible for the proposed implementation to use a profile routing functionality during the preprocessing phase, due to the fact that it is not available during the graph building phase in OTP itself. This has an obvious impact on the number of access-nodes and PNR-nodes found for each road
node, since the upper bounds used during the preprocessing phase are much higher than needed. We have seen in Table II how even an artificial decrease of the upper bounds can impact the size of the access-node and PNR-node sets. This indicates that the use of profile routing to find more accurate upper bounds during preprocessing could speed up the search by a great amount.

In addition to profile routing, other improvements can be made to speed up the search. As we showed in Figure 14 and Figure 15, bicycle parking locations are present in a great number, but they are not homogeneously spread throughout the city. It is actually the opposite, the bicycle parkings are placed mainly in groups at important locations (with some exceptions obviously). A natural approach to speed up two-way routing using a private bicycle as the initial mode of transportation could thus be to cluster bike racks that are placed very near to each other and treat them as a single bicycle parking area. This would decrease the number of total and per-node bicycle parkings, and thus also decrease the execution time of a single search. Lastly, a great improvement of the accuracy of the solutions would be to use real-time traffic information to delay the road arcs as needed and to have real-time public transportation information as well. Support for the latter is already present in OTP (although a real-time data feed needs to be made available by relevant transit agencies), while for real-time traffic information development is in progress but in an early stage.
APPENDIX

IMPLEMENTATION SETUP AND USAGE

In this chapter we discuss where to find and how to use the implementation that has been developed and presented in Chapter 5. We will describe how to retrieve the modified source code and how to setup the data needed for later use with the platform.

The implementation work has the form of extensions and changes to OpenTripPlanner (OTP)’s open source source code, thus for the setup and use of OTP in general and not of the extensions developed here we refer to its documentation[41] and webpage[32]

A.1 Usage

The source code has been “forked” on the source code hosting service GitHub[42] from the original source code repository[43] and is available at a separate location [44].

The preprocessing phase for the proposed algorithm is activated by adding the following parameters to the build.json settings file of OTP:

- ‘‘computeAccessNodes’’ : ‘‘true’’: to activate the access-node, PNR-node and bike PNR-node preprocessing during the graph building phase;
- ‘‘cityCenter’’ : ‘‘<lat>,<long>’’: the coordinates of the city center, if street direction computation is needed. This triggers the StreetDirectionModule to perform its work, as described in Section 5.2.1.
APPENDIX (Continued)

After the initial setup, performed following the original instructions, to use the newly added extensions some command line parameters for the routing platform have been added and are described as follows.

- **--printAccessNodes**: prints statistics and information about the computed graph’s access nodes, PNR-nodes and bike PNR-nodes;

- **--twoWayRouting**: uses the two-way PNR routing algorithm presented in Chapter 4 instead of the original A* variant as the main routing algorithm for requests;

- **--generateTestData**: generates a Comma Separated Value (CSV) file with random data; parameters for the generation are taken from the *generator.json* file, as described in Section A.2;

- **--testInput**: path to an input CSV file for the routing tests, as output by an execution using the **--generateTestData** command line parameter;

- **--testOutput**: path to an output CSV file for the routing tests;

- **--twoWayTest**: performs searches for every test example in the CSV input file using the two-way PNR routing algorithm and outputs a new CSV file containing both the inputs and the resulting durations of the trips, routing times and parking location coordinates;

- **--oneWayTest**: performs searches for every test example in the CSV output file of the two-way tests using the builtin variation of the A* routing algorithm and outputs a new CSV file containing both the inputs and the resulting durations of the trips, routing times and parking location coordinates;
A.2 Experiment example generation

We now want to give a brief description of the parameters that can be manipulated when generating experiment examples, on which to then test the algorithm. These parameters are specified in a file called `generator.json` that has to be in the same folder as the graph file to be used by OTP. The parameters that can be used by the generator procedure are the following:

- **"outputFile": "/path/to/output/directory"**
  sets the path where the output CSV file with all the information for the routing requests to be used in tests should be written;

- **"bboxSrc": "9.18,45.46,9.29,45.56"**
  represents the bounding box within which to randomly generate the source location of an experiment example. Represented as “minimum_longitude,minimum_latitude, maximum_longitude,maximum_latitude”;

- **"bboxSrcExcept": "9.18,45.46,9.23,45.49"**
  represents a bounding box within the source bounding box (bboxSrcExcept) that has to be ignored when randomly generating the source location of an experiment example. Represented as “minimum_longitude,minimum_latitude,maximum_longitude, maximum_latitude”;

- **"bboxTgt": "9.18,45.46,9.29,45.56"**
  represents the bounding box within which to randomly generate the target location of an
experiment example. Represented as “minimum_longitude, minimum_latitude,
maximum_longitude, maximum_latitude”;

- ‘‘arrFromTime’’: ‘‘0:00am’’
  starting time of the time window within which to randomly choose an arrival time at the
target location;

- ‘‘arrToTime’’: ‘‘11:59pm’’
  ending time of the time window within which to randomly choose an arrival time at the
target location;

- ‘‘depFromTime’’: ‘‘0:00am’’
  starting time of the time window within which to randomly choose a departure time from
the source location;

- ‘‘depToTime’’: ‘‘11:59pm’’
  ending time of the time window within which to randomly choose a departure time from
the source location;

- ‘‘experimentsNumber’’: ‘‘<N>’’
  number of experiment examples to generate.
CITED LITERATURE


VITA

Sergio Spinatelli

Education

| B.S., Engineering of Computing Systems |
| Politecnico di Milano |
| 2013 |

| M.S., Engineering of Computing Systems |
| Politecnico di Milano |
| 2015 |

| M.S., Computer Science (current) |
| University of Illinois at Chicago, Chicago, IL |
| August 2015 |