Solving time-dependent shortest path problems in a database context

Rodrigo Campi Sperb

March, 2010

Solving time-dependent shortest path problems in a database context

by

Rodrigo Campi Sperb

Thesis submitted to the International Institute for Geo-information Science and Earth Observation in partial fulfilment of the requirements for the degree in Master of Science in *Geoinformatics*.

INTERNATIONAL INSTITUTE FOR GEO-INFORMATION SCIENCE AND EARTH OBSERVATION

ENSCHEDE, THE NETHERLANDS

Disclaimer

Abstract

All around the world people experience delays due to bad traffic conditions. As a matter of fact, here in Netherlands, for instance, it is common sense to talk about the early morning pick time in the motorways and major roads just before people start their work duties. In this context, computation of shortest paths can no longer consider costs to traverse the network fixed, as traditionally, but rather as function of the time of the day. When these costs are known a priori (i.e., traffic condition information) the problem is usually referred to as time-dependent shortest path (TDSP). In this research two problem formulations are tackled: (1) a simple one-to-one TDSP for a given departure time (TDSP-GDT); and (2) One-to-one TDSP for a given interval of departure time (TDSP-LTT). A real-world time-dependent network from the Netherlands is used to test the developed solutions, and we implement the solutions in a database context. We make use of graph-theoretic approaches, more specifically an adaption of the famous Dijkstra's algorithm for static shortest path, proposed in the literature. Preliminary results indicate that TDSP-GDT problems can be solved at reasonable computational time when opportunities of optimizing the computation are taken by removing unnecessary nodes in the graph (i.e., trivial graph simplification). On the other hand, TDSP-LTT has shown to be very expensive computationally and only rather limited request sizes could be solved in a reasonable amount of time. That triggered us toward finding faster ways of solving the problem, with the concept of graph simplification being taken to other levels. We theoretically define a graph simplification to traverse dense subgraphs in which the computations are found to slow down. A proof-of-concept performance test shows that a careful delineation of dense subgraphs, brings much better results in runtime, also allowing a broader range of requests sizes to be solved, though the developed solutions may still have limitations when applied to (real-world) large graphs. At any case, our investigation serves as a basis upon which further work can be developed until fully operational solutions for TDSP problems in a database context can eventually be reached. In this note, we discuss other possibilities that could be exploited to achieve such a goal, particularly focusing on how to compute faster, speed-up techniques by precomputing parts of the problem, as well as pointing out that other graph simplifications can still be devised.

Keywords

dynamic shortest path, spatial database, time-dependent routing, graph-theoretic approach, graph simplification, speed-up technique

Acknowledgements

Praise be to God! O, Most High. Who was, is and will be. For all You have done in my life and taken me that far. You deserve to be praised and worshiped all of my days.

I want to express my most sincere appreciation to my supervisor Rolf de By for all the guidance and useful comments and discussion all the way of this research project. Also for the very good first impression of GFM coursework with Module 1 in Principles of Databases. From that moment I knew I had to do something relate to this subject.

I want also to thank my second supervisor Otto Huisman for the few but useful discussions during the work. As well as to all my colleagues at GFM course that made life at research period still enjoyable, and for also helping one and another with discussion and tips related to our projects. In this note, I am specially grateful for the partners in SDIT research group, particularly Adam (Runner!) and Shelton (P!) for all shared experiences, tips and etc.

I am grateful to NUFFIC for funding my studies and this wonderful opportunity of living abroad. And to all administrative staff and lectures in ITC for all the help throughout my period of study.

I can't help thanking the outstanding support that my family always has given to me. I excuse myself for some lines in my mother tongue directed to them: *Pai e mãe, eu lhes agredeço por tudo que fizeram por mim toda a minha e vida, e por todo o apoio neste período longe de casa. Eu os amo muito, de todo meu coração. Estendo a toda minha família por todo o carinho, amor e consideração que sempre tiveram comigo. Irmãos, tios, primos, avós e bisa, amo a todos.*.

God has richly blessed me in the personal sense here too. I now excuse myself once more for some words in another language: *Yely, mi amor lindo, gracias por todo el apoyo, sonrisas, abrazos y cariño todo este tiempo. Te amo muchototote*.

I finally commit this document into Your Holy Hands, my dear Lord.

Rodrigo Sperb
Enschede, February 2010.

Contents

Abstract **i**

Acknowledgements **iii**

List of Tables **vii**

List of Figures **ix**

1 Introduction **1**
- 1.1 Motivation and problem statement 1
- 1.2 Research identification 3
  - 1.2.1 Research objectives 3
  - 1.2.2 Research questions 3
  - 1.2.3 Innovation aimed at 4
- 1.3 Method adopted 4
- 1.4 Thesis outline 5

2 Dynamics in Vehicle Routing **7**
- 2.1 Why Dynamic? 7
- 2.2 Generating time-varying travel times 8
  - 2.2.1 Travel times/Speed profiles—a good approximation . . . 12
- 2.3 The role of Shortest Path Problems 13

3 Dynamic Shortest Path approaches **15**
- 3.1 Dynamic Shortest Path problems 15
- 3.2 Dynamic Shortest Path solutions 19
  - 3.2.1 The Discrete-time algorithm 19
  - 3.2.2 The Bellman-Ford based algorithm 20
  - 3.2.3 The adapted A* algorithm 21
  - 3.2.4 The Two-step time-dependent shortest path algorithm . 22
- 3.3 Speed-up techniques for shortest path computation 24
  - 3.3.1 Goal-oriented routing 25
  - 3.3.2 Hierarchy-based routing 29
  - 3.3.3 Combination of speed-up techniques 32
  - 3.3.4 Techniques for a dynamic setting 32
- 3.4 Dynamic Shortest Path in a database context 34

4 Tools for Dynamic Shortest Paths in a database context 35

4.1 A set of Time-dependent Shortest Path (TDSP) problems and definition . . . 35

4.1.1 A simple one-to-one TDSP for a given departure time . . 37

4.1.2 One-to-one TDSP for a given interval of departure time . 37

4.2 Relevant concepts and definitions for solving TDSP problems . 37

4.3 Solving a set of TDSP problems . . . 39

4.3.1 Explaining the Two-step LTT approach . . . 40

4.3.2 Identifying tools to be implemented for the Two-step LTT approach . . . 42

4.4 Auxiliary tools for the Two-step LTT approach . . . 43

4.4.1 Initialization tool: generate initial arrival-time functions 44

4.4.2 Tools for X-/Y-value look-up . . . 45

4.4.3 Tool to add-up two mathematical functions . . . 46

4.4.4 Tool to determine the minimal function from two mathematical functions . . . 47

4.5 Two-step LTT approach for solving a set of TDSP problems . . . 48

4.6 Interlude: encompassing multimodal shortest path problems—a conceptual level discussion . . . 48

4.6.1 Why only at conceptual level? . . . 48

4.6.2 Multimodal TDSP in a single mode TDSP setting . . . 48

5 Case study implementation and performance tests 53

5.1 A time-dependent GIS dataset of the Netherlands . . . 53

5.1.1 Characteristics of the transportation network . . . 54

5.1.2 Loading the Netherlands time-dependent network dataset into a database . . . 57

5.2 Abstracting a Graph for applying (time-dependent) shortest path algorithms . . . 58

5.3 Delivering edge-delay functions from speed profiles . . . 61

5.4 Solving Time-dependent Shortest Path for a Given Departure Time (TDSP-GDT) in a database context with generalized Two-step LTT 62

5.4.1 Preliminary computational results: towards fast computation . . . 63

5.4.2 Graph simplification—or how to get faster TDSP-GDT problem results . . . 65

5.5 Solving Time-dependent Shortest Path for a given interval of departure time (Least Travel Time) (TDSP-LTT) in a database context with Two-step LTT . . . 68

5.5.1 Preliminary computational results: finding limitations . 69

5.6 Revisiting the graph simplification—or how to quickly traverse a dense part of the graph . . . 71

5.6.1 Defining a dense subgraph to substitute for the simplified graph . . . 73

5.6.2 Precomputing least-delay functions for the simplified graph 74

5.6.3 Finding the actual path through a dense subgraph from least-delay functions . . . 76

5.6.4 Substituting dense subgraphs by simplified subgraphs . 77
5.6.5 Proof-of-concept speed-up performance check 77

6 Discussion, Conclusions and Recommendations 83
6.1 Outline of the results and discussion 83
6.1.1 Further discussion—Looking outside the box 89
6.2 Conclusions 90
6.3 Recommendations 91

Bibliography 93

A Preparation of dataset for case study 101
A.1 Loading GIS dataset into the database 101
A.2 Loading time-dependency of the road segments 101
A.3 Abstracting the graph 102

B Solving TDSP problems in a database context 105
B.1 Auxiliary tools 105
B.1.1 Initialization tool—TDSP-GDT 105
B.1.2 Initialization tool—TDSP-LTT 107
B.1.3 Look-up tools 108
B.1.4 Update arrival-time functions tool 109
B.2 Two-step LTT for TDSP-GDT 110
B.2.1 Time-refinement step—TDSP-GDT 110
B.2.2 Path-selection step—TDSP-GDT 110
B.2.3 Shortest-path request—TDSP-GDT 111
B.3 Two-step LTT for TDSP-LTT 112
B.3.1 Auxiliary tools 112
B.3.2 Time-refinement step—TDSP-LTT 113
B.3.3 Path-selection step—TDSP-LTT 115
B.3.4 Shortest-path request—TDSP-LTT 116

C Optimization procedures 117
C.1 Sequence of commands for the trivial graph simplification . . . 117
C.2 Generating aggregated edge-delay functions 118
C.2.1 Aggregated edge-delay function algorithm 120
C.3 Sequence of commands for the dense subgraph simplification . 120
C.4 Identify entry nodes and exit nodes 120
C.4.1 Precomputation of least-delay functions and graph simplification 121
C.4.2 Average performance tests 126

List of Tables

2.1 Reported gains with integration of time-varying travel times in vehicle routing. 11
2.2 Findings of studies applying FCD to estimate traffic parameters. 12

3.1 Reduction to two fundamental problem variants and methods of computation [21]. 19
3.2 Classification of TDSP problems after [15]. 20

4.1 Dimensions on TDSP problem requests. 36
4.2 Summary of tools for the development of the Two-step LTT approach. 45

5.1 Distribution of direction of modeled segment by hierarchy in the Tele Atlas dataset roads network. 57
5.2 Distribution of time-dependency in the Tele Atlas dataset roads network. 57
5.3 Summary of steps carried out for loading the dataset into the database. 59
5.4 Reduction in the number of nodes by applying the simplification of graph with gradual inclusion of other levels of hierarchy levels of roads. 68
5.5 Precomputation time for substituting dense subgraphs by simplified version with least-delay functions to allow quick traversal of these dense subgraphs—first attempt. 79
5.6 Impression on speed-up achieved by the graph simplification in varying distance between start and end node (*interval for departure time refinement of 10 min). 79
5.7 Precomputation time for substituting dense subgraphs by simplified versions with least-delay functions to allow quick traversal of these dense subgraphs—second attempt. 80
5.8 Impression on computational performance after graph simplification to allow quick traversal of dense subgraphs, with varying interval for departure time (*distance between start and target node of +120 km). 81
5.9 Average response over 20 requests with randomly varying distances between start and target node, and fixed time intervals for departure. 82

List of Figures

1.1 Investigated setting with databases playing a role also as environment for computation of routes 2

2.1 An architecture for a real-time decision system for vehicle routing problem (after [81]). 9
2.2 An example of structure and environment of a Dynamic Routing System (after [32]). 10
2.3 Smoothed travel time function [31]. 13

3.1 Definition and an example of a graph (adapted from [16].) . . . 16
3.2 Dijkstra's algorithm (after [85]). 17
3.3 Orda Rom algorithm ([66] after Ding et al. [28]) 21
3.4 Outline of Two-step least travel-time algorithm [28] 22
3.5 Ding et al. (2008) [28] time refinement algorithm. 23
3.6 Ding et al. (2008) [28] path selection algorithm. 24
3.7 An overview of shortest path algorithms and speed-up heuristics (adapted from [60]). 26
3.8 Landmarks (l_1, l_2) and triangle inequalities intuition [24]. . . . 27
3.9 Idealized shape of the search space for shortest path computations in different methods (after [60]). 28
3.10 Preprocessing connections from a cluster V_i [60]. 29
3.11 Intuition of identifying a highway edge. 31

4.1 TDSP problem formulations in increasing complexity levels. . . 36
4.2 Intuition and example of relevant concepts for solving TDSP problems. 38
4.3 Outline of Two-step least travel-time algorithm [28] 41
4.4 Ding et al. (2008) [28] time refinement algorithm. 42
4.5 Running example of updated of arrival-time function. 43
4.6 Ding et al. (2008) [28] path selection algorithm. 44
4.7 Running example of an optimal path-selection p^* for the optimal starting time t^*. 44
4.8 Initialization algorithm. 46
4.9 Look-up algorithms. 46
4.10 Add-up two functions algorithm. 47
4.11 Determination of the minimal function algorithm. 47

4.12 Two-step LTT approach [28] with adapted presentation to include auxiliary tools developed. 49
4.13 Two-step LTT approach [28] generalized to solve TDSP-GDT. . 50
4.14 Allowing extra time for mode changes in a multimodal network by creating a "virtual" edge that connects the switch point to itself before reaching the subsequent node of a different mode. 51
4.15 Representing time-windows in the edge-delay function of the transition of modes in a multimodal network. 52

5.1 Snapshot of part of a speed profile data from Tele Atlas dataset. 54
5.2 Tele Atlas dataset link between speed profiles and transportation network segments (adapted from [6]). 55
5.3 Polynomial fit of the 60 speed profiles for one product tile of the Tele Atlas dataset [7]. 55
5.4 Tele Atlas transportation network dataset 56
5.5 Time-dependent segments in the Tele Atlas dataset roads network.) 58
5.6 Abstraction of the graph in the database. 60
5.7 Conceptual model of the time-dependent graph. 61
5.8 Overview algorithm for getting profile with inverted speed values. 62
5.9 Derivation of edge-delay functions from inverted relative speed profile. 62
5.10 First impression on performance for TDSP-GDT solution in a database context for the case study. 64
5.11 First impression on performance for TDSP-GDT solution in a database context for the case study, after creating indexes to constantly used table columns as well as precomputing edge-delay functions to the graph. 65
5.12 A snapshot with an impression on unnecessary nodes in the graph that could be "dissolved". 66
5.13 Dissolve edges algorithm. 67
5.14 Subset of the graph to which the simplification of the graph was applied. 67
5.15 Sum-up edge-delay functions of simplified edges algorithm. . . . 68
5.16 First impression of speeding-up the performance of TDSP-GDT solution in a database context for the case study, after simplifying the graph and generating aggregate edge-delay functions. . . . 69
5.17 First impression of the performance of TDSP-LTT solution in a database context for the case study, after simplifying the graph and generating aggregate edge-delay functions. 70
5.18 Computational time to update arrival-time functions as the iterations go on. 71
5.19 Intuition of the substitution by a simplified graph in a dense subgraph. 72
5.20 Conceptual model of the substitution by a simplified graph in a dense subgraph. 73
5.21 Define the dense subgraphs G_k. 74

5.22 Identify entry nodes of a tile S. 75
5.23 Identify exit nodes of a tile S. 75
5.24 Computing least-delay functions for the dense subgraph inside a tile S. 75
5.25 Add-up two edge-delay functions algorithm. 76
5.26 Getting the minimal of two functions tool algorithm. 76
5.27 Substitution of the dense subgraph for a simplified version algorithm. 77
5.28 Arbitrarily defined dense subgraph to be simplified—first attempt. 78
5.29 Arbitrarily defined dense subgraph to be simplified—second attempt. 80
5.30 Runtime complexity in relation to the size of the subgraph when applying the graph simplification. 81

List of Acronyms

TDSP	Time-dependent Shortest Path
TDSP-GDT	Time-dependent Shortest Path for a Given Departure Time
TDSP-LTT	Time-dependent Shortest Path for a given interval of departure time (Least Travel Time)
VRP	Vehicle Routing Problem
DVRP	Dynamic Vehicle Routing Problem
ICT	Information and Communication Technology
FCD	Floating Car Data

Chapter 1

Introduction

1.1 Motivation and problem statement

Definition of routes is of major concern in logistics and transportation. The way computing technology can aid the definition of routes is through computational solutions that minimize costs for taking goods from one place to another. This optimization task may take different degrees of complexity, from the simplest definition of shortest path routes to a variety of more well-defined complex problems generically called vehicle routing problems. For the sake of clarity, in our research context routing problems are only those that seek for shortest paths, whatever are the costs associated to the links.

Most traditional approaches make use of distance or a distance-dominated function as cost to be minimized, as it is in the problem introduced by Dantzig and Ramster in 1959 [19]. When that is the case, a shortest path algorithm like the good one of Dijkstra [16] suffice to find a solution of the problem. Over the years, finding the shortest path over a network is an issue that has received quite some attention from research. These efforts have produced a number of solutions, either based on exact algorithms or some heuristic,[1] as well as empirical findings on computational performance [90]. In most cases, these solutions are based on graph-theoretic approaches, in which the segments that connect the nodes of a network have a cost associated, usually just distance or a distance-dominated cost function. The task of the algorithm is to find a set of segments that connect origin to destination while minimizing the cost.

This traditional approach of shortest path determination may accommodate many applications (e.g., specific situations in which there is no arrival of dynamic requests and the dynamic changing network conditions do not interfere for finding the solution); but fail to deal with problems in which movement is time-dependent [47]. In many real-world applications, this should not be neglected, as stated in [54] "once stochastic and/or dynamic information becomes certain even while vehicles are en route". This has brought a whole new perspective to routing problems, generically defined as *dynamic shortest paths* or

[1]Heuristic method is one used to rapidly find the best solution possible, which eventually may not be the optimal one. In this case, there is a trade-off between computational expenditure and optimality, whereas a near-optimal solution may be considered to be enough for providing quick answers.

dynamic routing. Under this perspective, relevant information may be available even on a real-time basis, which has received attention from Operational Research since the last decade [81]. The main focus of this perspective is the realization that factors may interfere in the travel time along the route, which eventually can cause the shortest path to not be the preferable one if travel times based on dynamically changing parameters are considered for defining the route.

In addition to the complexity of having varying travel times, a realistic logistics and transportation setting may also involve multiple means of transportation (see [74]), as well as others aspects that can turn out the routing problem more and more complex. By more complex we mean to have to consider more variables that eventually will influence the determination of the best (shortest) path from moving to one place to another. All those aspects presented may be part of a *'puzzle'* for defining a realistic approach for realizing shortest paths, for instance to be used in vehicle routing problems, thus aiding decision-making for logistics and transportation. They may still be fairly solvable with graph-theoretic approaches [47, 15, 40]; that is because graphs has been for long proven to be a mathematical abstraction that suits many situation and problems [16]. What's more, a still actual call for Intelligent Transportation Systems (ITS) demands *information technology* [41], that today relies on database as means to structure, organize and retrieve data, as well as to compute solutions and hence be at the base of such system implementation. Here it is important to notice this last argument, as today's practice envisions databases working within the concept of object-oriented encapsulation, with all the functions a user may require to interact with data being provided on the server side (Figure 1.1). Therefore, an investigation on solving shortest paths for dynamic shortest path problems in a database context, as intended by this research, is believed to be in line with research in the field. From the best of our knowledge, we may refer to the work on finding time-dependent over large graphs in a spatial temporal database [28], that only treats a single kind of time-dependent shortest path problem and have database simply as mean of storing the data; and the work on shortest path for moving objects in spatial network databases [87], which may be helpful for dealing with a problem formulation in which the route should be defined while the vehicle is en route.

1.2 Research identification

1.2.1 Research objectives

The main goal of this research is to carry out an investigation of solving dynamic shortest path problems in a database context. To accomplish that, three dimensions are identified as necessary to be dealt with:

Problem complexity refers to the intrinsic complexity of well-defined dynamic routing problems and their materialization in terms of *computational complexity*, in *time* and *space complexity* measures.

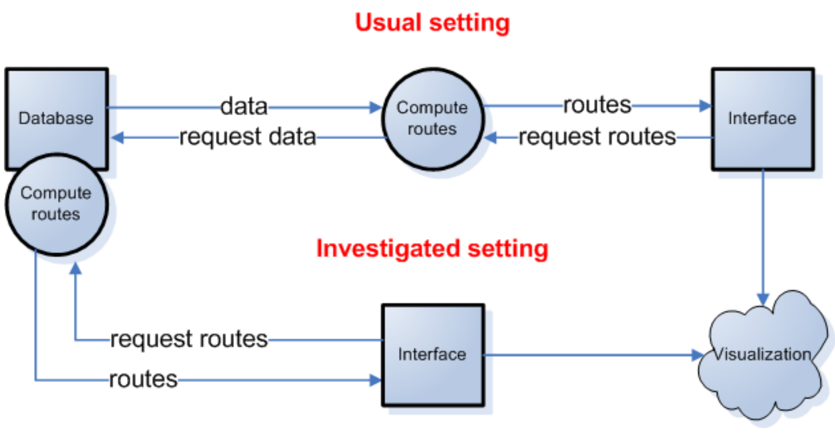


Figure 1.1: Investigated setting with databases playing a role also as environment for computation of routes

Data modeling is about modeling real-world dynamic problems and their complexity in a database context, and properly translating this complexity into a form computation of shortest paths with known approaches are possible.

Performance as an indication of how well computation of routes will perform in the database context, for the well-defined dynamic routing problems aforementioned.

As consequence, the following *defacto* objectives for the research are defined:

- Identify data model requirements for the time-dependent real-world dataset, and then materialize (i.e. translate) those in a form that allows computation of (shortest path) routes, with known approaches.
- Develop algorithmic solutions for well-defined dynamic routing problems with different complexity levels in a database context.
- Test the solutions against different sizes of dataset to get insight in the complexity in different levels of problem.
- Set up experiments to indicate performance of the developed algorithms in the database context.
- Present approaches to occasionally speed up the computation of shortest paths against the real-world dataset.

1.2.2 Research questions

From the research objectives we can define a set of questions to be answered by the investigation as follows:

1. How can data modeling properly translate real-world problems and dataset into a form that the implemented solutions can work with?

2. Can different degrees of complexity in dynamic routing solutions all be effectively and efficiently implemented in a database context?

3. What are the limitations on computational complexity of the developed solutions?

4. How can performance of the implemented solutions for a real-world problem dataset be enhanced?

1.2.3 Innovation aimed at

Computation of shortest paths in a dynamic problem setting is not *per se* an innovation, as several approaches have indeed been proposed to solve different problems and situations. Nevertheless, it still not known a previous work that conduct an investigation with approach similar to the one here proposed. Especially in relation to the database context, chosen for this research, and focus on providing solutions for different problem formulations. It is fair to mention that initiatives of solving shortest path problems in a database context do exist [28, 87], but those treat only one kind of routing problem, not a variety of them, as intended in this research, as well as not always consider solutions that really run in a database context. We also could mention one initiative that is even open source and freely available (pgRouting[2]), for an also open source database management system (PostgreSQL), but hitherto they only work in a setting that is intrinsically static.

Finally, our research also has a focus on performance issues, and on how a proper use of data modeling can eventually help to deliver request responses in a faster way. By that we mean not only to apply the solution developed on the dataset and report their performance, but to reason over preliminary results and somehow come up with solutions to eventually speed up the computation of dynamic shortest paths in a database context.

1.3 Method adopted

Dynamic shortest path determination is a fundamental part of dynamic vehicle routing problems, just as shortest path is for the archetypical (static) Vehicle Routing Problem (VRP), though in logistics and transportation other parameters may need to be optimized as well [72] (e.g., fleet size, dispatching schema). As our focus is on the dynamic shortest path computation and on an investigation on the computational complexity of different levels of the problem, addressing as well as data model and performance issues, we can define the steps for attaining the research objectives as follows:

1. Define a set of gradually increasing complexity levels of dynamic shortest path problems and properly formulate them, to be implemented in a database context.

[2]`http://pgrouting.postlbs.org/`

2. Search in literature available mathematic or algorithmic approaches solutions for the defined set of problems formulation and implement the solutions in the chosen database context.

3. Address the pertinent data model issues to properly convey required arguments for the implemented algorithmic solutions.

4. Test the implemented algorithmic solutions against a real-world and dynamic changing network dataset and get insight in their performance for solving requests of routes.

5. Define a set of scalable request sizes to evaluate performance issues related to scalability of requests—here, some attempt may be drawn to enhance performance.

The steps taken were not necessarily conducted in a sequential way as presented, nevertheless this should provide a *picture* of how the research was carried out. We detail the activities of each step, focusing on the three dimensions of the research below.

Problem complexity: to support the definition of the set of dynamic shortest path problem formulations to be solved in the database context, a comprehensive search in literature for dynamic shortest paths was carried out. This *literature review* also provided promising approaches and algorithmic solutions published in scientific sources.

Data modeling: for the set of different complexity problems formulation, data modeling issues were addressed, particularly focusing on how to translate the available real-world dataset so that known and proved approaches may be applied in a database context.

Performance: dynamic shortest path problems are likely to require solutions that perform reasonably fast, sometimes even in real-time, if we think of defining routes while a vehicle is en route. We tested the algorithms implemented in the database context against real-world dataset, and particularly focus on getting insights on enhancing the performance of the solutions developed.

1.4 Thesis outline

This thesis has been divided into six chapters:

Chapter 1 presents the motivation for this research project and states its problem, objectives and questions, as well as describes the adopted method.

Chapter 2 refers to an application in which the subject of this research project is believed to play a major role, namely Vehicle Routing Problems, but mainly point out to the need for considering the dynamics of traffic conditions when computing routes. The chapter discusses issues related to that, as well as presents reported gains found in the literature when variations in traffic conditions are explicitly used in routing applications, bringing the concept of dynamic shortest paths.

Chapter 3 provides an overview of what is a shortest path problem, defines the static and dynamic counterparts, to then provide approaches found in the literature to solve this kind of problem. Additionally, a discussion on existing speed-up techniques for fast computation of routes is given. The chapter finalizes by briefly stating why we consider the solution of shortest path problems in a database context.

Chapter 4 starts with the formulation of the (time-dependent shortest path) problems tackled in the project, followed by an introduction of relevant concepts for solving them. From this point on, we present and explain the algorithmic approach of the chosen solution, including auxiliary tools that need to be implemented along with it. All of this is in done as much as possible in a implementation-independent form. Finally, an interlude with a conceptual discussion on encompassing multimodal routing into a single mode setting is also presented.

Chapter 5 is the implementation of the solutions introduced in Chapter 4 regarding a case study with a (real-world) time-dependent network dataset from the Netherlands. Performance of the solutions are tested and the limitations are identified. That triggered an exercise on optimizing the solutions towards faster computation of routes which is also presented in this chapter.

Chapter 6 outlines the results obtained in the research project also putting them in context of the research objectives and questions designed for the project. Further discussion on the subject in its broader sense is also provided, followed by the conclusions drawn for the project. Recommendations for further improvements are also presented.

Chapter 2

Dynamics in Vehicle Routing

2.1 Why Dynamic?

The traditional VRP is about minimizing the total cost of a distribution system [14], as it was introduced by Dantzig and Ramster in 1959 [19]. In most classical models and techniques (i.e., static), the costs associated to traverse from one point to another were distance-dominated functions, that do not capture "essential trade-offs needed to understand and effectively operate transportation systems, as indeed it is more important to minimize delivery times" [14]. This essential character of time [71], with the inclusion of dynamic elements [14], would be much better captured in a dynamic fashion of such problems.[1]

Among the very first to consider the dynamic version of vehicle routing problems [58], Psaraftis [71] stated that Dynamic Vehicle Routing Problem (DVRP) traditionally means dispatching of services that evolve in a real-time, i.e. *dynamic*, basis. This is specially relevant nowadays"“where the world's economies are more and more interdependent" [72], bringing a whole need for enhancing the efficiency of distribution systems; and what basically distinguishes a dynamic vehicle routing problem from a *static* one is [71]:

- *"if the output is not a set of routes, but rather a policy that prescribes how the routes should evolve as a function of those inputs that evolve in real-time."*

Following Psaraftis definitions, Larsen (2000) [53] formulated his definition of a VRP having the nature that:

- "Not all the information relevant to the planning of the routes is known by the planner when the routing process begins."
- "Information can change after the initial routes have been constructed."

Some important characteristics that differentiate dynamic from static vehicle routing problems were listed by Psaraftis [71]. In summary, the author

[1]One application that would greatly benefits from routing that minimizes travel times is delivery of perishable food (e.g., [67, 45]).

recognizes the essential role of the time dimension, but also point at the issues of uncertainty and update due to incoming information, and the need for high performance.

Approaching VRP in a dynamic fashion has recently gained more attention [81], particularly with the advent of revolutionary Information and Communication Technology (ICT) [72, 41, 32, 44], "that allow dynamic information to become available even while vehicles are en route" [54]. New technological advancements allows to effectively process data in real-time, which is a kind of distribution system that is to become the "norm in the future" [72].

Fleischmann et al. [32, 31] state that "in the majority of literature, the only dynamic element is the arrival of customer orders during the planning period"—as in the likely scenarios of real-time distribution mentioned in [72]. But that is not the only dynamic element that VRP should address or incorporate [14], as time plays an essential role [71], especially in the sense of conditions of traffic—i.e., travel times [14] for moving from one place to another. Malandraki and Danski (1992) [59] recognized the importance of considering that travel times may vary along the day, and defined a *time-dependent* VRP—essentially only having dynamics in travel times— as the counterpart of a DVRP in its classical approach—i.e., dynamics of customer orders arrival [32, 31]. To consider fluctuation of travel times across the network edges is then a main issue. Indeed, the importance of traffic conditions turns out to be clear if we consider that in the U.S. urban areas travel delays totaled 4.2 billion hours for the year of 2007 [78]. In contrast, the simple inclusion of the dynamics of travel times has been reported to enhance efficiency of different distribution systems (e.g., incident response management [46], electrical goods wholesaler [57], that also reported savings in CO_2 emissions in 7% as an additional finding); or to better approximate real travel times against constant travel times, that was reported to be underestimated by 10% in average true travel times in Berlin [31] and to realize a driving time advantage between 5 to 15% in the motorways at conurbations Rhine-Ruhr and Rhine-Main (Germany) [43] (see Section 2.2 for more on that).

To summarize the idea of dynamically routing vehicles and managing distribution systems, Séguin et al. [81] presented a high-level architecture of a real-time decision system for dispatching problems (Figure 2.1). In their architecture, the environment (including new order requests and traffic conditions) are monitored by the *characterizer*, used for predictions and estimations by the *projector*, until eventually an action plan is generated in the *effector*, after the *planner* evaluates possible alternatives of reactions. Determination of optimal routes would then be a part of that *planner* component of the system. As we are particularly interested in the dynamic element of conditions of traffic, we provide a discussion on generating dynamic travel times for a network below. Though some authors restrict the definition of DVRP for only where orders arrive during the planning horizon and decisions are made in a real-time basis [35, 31], for our purpose we only consider the variation of travel times as a *dynamic element*, which should be incorporated to better capture real-world phenomena in vehicle routing. At any case, making use of traffic monitoring information for computation of (shortest) routes on request would certainly be

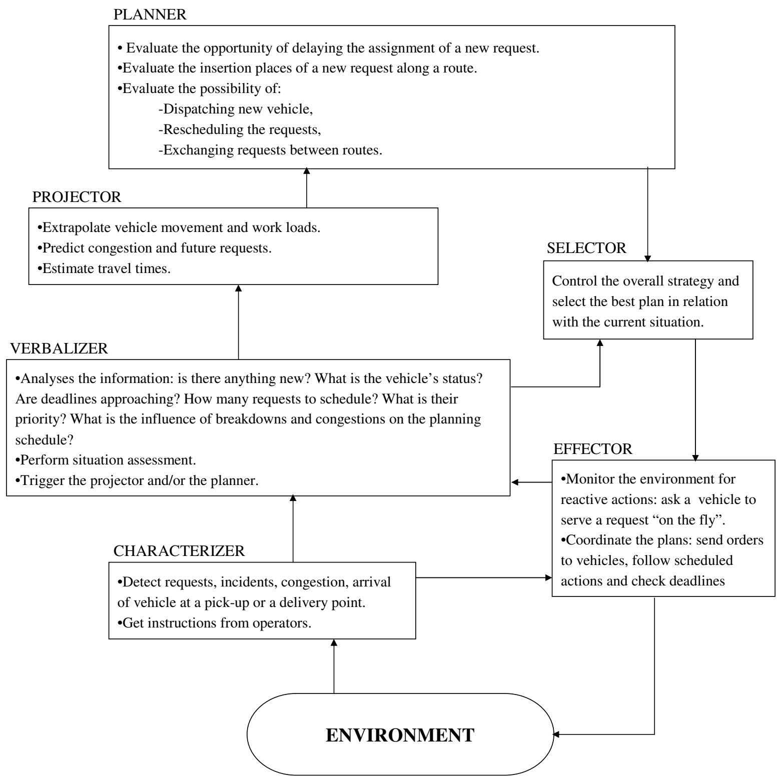

PLANNER
• Evaluate the opportunity of delaying the assignment of a new request.
•Evaluate the insertion places of a new request along a route.
•Evaluate the possibility of:
-Dispatching new vehicle,
-Rescheduling the requests,
-Exchanging requests between routes.
PROJECTOR
•Extrapolate vehicle movement and work loads.
•Predict congestion and future requests.
•Estimate travel times.
SELECTOR
Control the overall strategy and select the best plan in relation with the current situation.
VERBALIZER
•Analyses the information: is there anything new? What is the vehicle's status? Are deadlines approaching? How many requests to schedule? What is their priority? What is the influence of breakdowns and congestions on the planning schedule?
•Perform situation assessment.
•Trigger the projector and/or the planner.
EFFECTOR
•Monitor the environment for reactive actions: ask a vehicle to serve a request "on the fly".
•Coordinate the plans: send orders to vehicles, follow scheduled actions and check deadlines
CHARACTERIZER
•Detect requests, incidents, congestion, arrival of vehicle at a pick-up or a delivery point.
•Get instructions from operators.
ENVIRONMENT

Figure 2.1: An architecture for a real-time decision system for vehicle routing problem (after (81)).

part of any dynamic routing system.

2.2 Generating time-varying travel times

Travel time data is essential for routing vehicles, "both for modeling time constraints (e.g., time windows, time objectives) as well as for minimization of travel time, waiting time, etc." [31]. Vehicle routing algorithms, however, have traditionally been developed under the average-speed-for-links paradigm [57]. Even when studies refer to a dynamic vehicle routing perspective, what they truly mean is often arrival of orders in a dynamic fashion [72, 71, 32, 31]. This can be partially explained by the difficulties related to generating sufficient coverage of traffic conditions data [31, 89] and integrating traffic flow parameters into routing systems—as pointed out by [14]—but also by limitations of the algorithms themselves, "that in many cases were not drawn to make use of time-varying travel times" [31]. Yoon et al. [89], for instance, point out the limitations in coverage for monitoring traffic as the network gets denser. Still, if enough traffic condition data can be generated, it remain issues of how to adequately integrate them to a routing system in a dynamic basis, i.e. considering them as travel costs[2] that vary over time, as they may need to be processed beforehand.

Recent technological advancements, however, have much to contribute to overcome some of those issues [34]. As an example, Helling & Schoenharting [43] state that traffic information is now available in most European countries via the RDS/TMC radio channel .[3] Fleischmann et al. [32] present a planning framework that integrates dynamic travel times into a dynamic routing system, with online communication with drivers and customers also in place (Figure 2.2). Huang & Pan [46] proposed an "incident response management tool" (IRMT) that couples GIS with traffic simulation to optimize multiple incident response, also reporting a case study to test and evaluate the proposed tool with real-world data from the Clementi area of Singapore. They have in common the ability of allowing the integration of costs of traverse that vary over time, sometimes even in a real-time basis. Table 2.1 summarizes some practical advantages found in the literature with the inclusion of time-varying travel costs.

Unfortunately, technology alone can not guarantee that relevant information on travel costs fluctuation is always available, particularly when it comes to quality of traffic parameters dynamically generated. We refer to Helling & Schoenharting [43] for more on quality of dynamic traffic information. But even if information of good quality is generated, it has to be combined with affordable methods of estimating dynamically real time travel costs, but also with reliable and efficient—i.e., reasonable computational complexity—methods to provide time-varying travel costs in a good basis. For the former, the use of so-called Floating Car Data (FCD) emerges as a promising option (see Table 2.2) [31, 89, 68]; that means, to make use of on traffic vehicle themselves to generate information from which they may benefit. For the latter, see [55] for a good description of methods to estimate travel times, as well as a proposed new approach for doing it on long freeway sections. We also point at the approach of van Woensel et al. [84] which claims to outperform others usually applied in the simulation of the time-dependency related to the inclusion of traffic congestion in vehicle routing. Also the method of Yoon et al. [89] provides a good basis on how to process FCD data and generate dynamic traffic information. Moreover, the findings of Fleischmann et al. [31] are also relevant, as they show that a few time slots of time-varying travel costs are already good enough to provide excellent approximations of true travel costs. Finally, Pfoser et al. [69] recognize the need for an accurate and reliable weight database (i.e., some travel time related parameter to reflect conditions of traffic) as the basis for a dynamic navigation system, with the use of map-matching algorithm to dynamically associate costs to the network [13] and data management techniques for delivering dynamic weights from FCD. To overcome shortcomes with eventual lack of online data, Leonhardt [55] points out the combination with a historical database of traffic conditions as a good direction. He presents approaches to derive travel time

[2]Though travel costs is a generic term that may encompass a number of somewhat different kinds of costs, we here use travel costs as a synonymous of travel times.

[3]RDS/TMC is an international standard for delivery of traffic information to navigation systems.

Table 2.1: Reported gains with integration of time-varying travel times in vehicle routing.

What?	What about?	Who?
Effect of using constant average and time-varying travel times with different numbers of time slots. Data from Berlin, Germany.	Constant average times lead to approximately 10% of underestimation of the true travel times. For 5-10 times slots excellent approximation of the true travel times may already be obtained, though use of more slots do not increase significantly computational complexity.	Fleischmann et al. [31]
Comparison of existing dynamic routing guidance, an ideal one (better quality of information) and static routing guidance for commuting trips between Duesseldorf and Cologne, Germany.	Existing dynamic routing guidance were found to allow only 5% of driving time advantage, while the idealized system, with optimal message quality assumption, should lead to an improvement of around 11%. Shortcomings related to quality of information and availability of infrastructure (see below for details on that).	Helling & Schoenharting [43]
Coupling of GIS and traffic simulation for an incident response management tool (IRMT). Test case in Singapore.	Approach led to 10-25% of reduction of travel times compared to conventional dispatching strategy.	Huang & Pan [46]
Use of a queueing approach to incorporate time-dependency of travel times. Comparison of the approach is done against time-independent and three time zones approaches.	Explicitly make use of the time-dependent congestion results in routes that are quite considerably shorter in terms of travel times. Extra calculation time needed to process large dataset is worthwhile as the solution quality greatly improves.	van Woensel et al. [84]

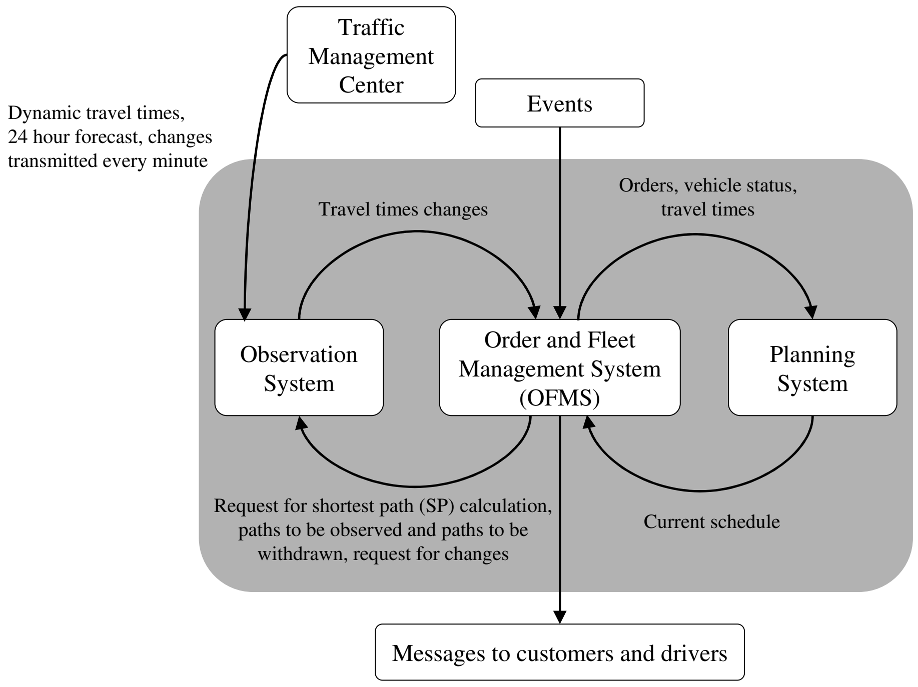

Traffic Management Center
Events
Dynamic travel times, 24 hour forecast, changes transmitted every minute
Travel times changes
Orders, vehicle status, travel times
Observation System
Order and Fleet Management System (OFMS)
Planning System
Request for shortest path (SP) calculation, paths to be observed and paths to be withdrawn, request for changes
Current schedule
Messages to customers and drivers

Figure 2.2: An example of structure and environment of a Dynamic Routing System (after (32)).

data from online measurement, as well as to estimate this in the absence of online data, and to predict travel time with the use of regression techniques combining measured online data with a historical database of traffic condition.

Though both recent technological and methodological advancements related to considering the time-dependency of travel costs in vehicle routing problems have shown that they do provide advantages in terms of actual travel costs, it still is challenging to do so. For example, Fleischmann et al. [31] comment that traffic information is generated in a too high level of detail in traffic information systems to be used straightaway in computing travel times matrices (usual transportation routing approach). It would cause unnecessary effort and memory requirements. That is why they have investigated what are the effects of aggregating traffic information on time slots (see Table 2.1). In contrast, Helling & Schoenharting [43] do not see limitations on integrating dynamically generated traffic information even for en route dynamic route guidance computation; but in their study case, they also show that we still struggle with the quality of traffic messages transmitted, which in average were approximately correct only in 35% of the transmitted travel costs. Accordingly to the authors, that limits the potential benefits of dynamic route guidance, as do so the infrastructural lack of alternatives routes in a practical point of view, on the other hand.

Table 2.2: Findings of studies applying FCD to estimate traffic parameters.

Description	Author(s)
Defines algorithms and data management techniques for delivering a dynamic traffic weights database for routing over so-called Dynamic Travel Time Maps.	Pfoser et al. [69]
Uses a map matching algorithm to dynamically attach travel times collected by FCD in a network (no check on the quality of the information generated, though).	Pfoser et al. [68]
Results shows that used approach allowed an accuracy of 90% in characterization of traffic parameters, thus the use of a longer history would permit identifying traffic conditions with even higher accuracy.	Yoon et al. [89]

2.2.1 Travel times/Speed profiles—a good approximation

With all existing technologies for monitoring traffic conditions as well as emerging ones like FCD, that do allow to generate traffic information on a real-time basis, it might look tempting to straightaway use them for computing routes in a dynamic vehicle routing system perspective. As a matter of fact, in situations of stochastic event, such as accidents, road interruption and others may still occur, indeed a system that delivers information on traffic condition en route can be suggested as a measure towards alleviating problems with congestion [34]. Under stochastic conditions, Nie & Wu (2009) [65] study the finding of *a priori* shortest path as “reliable paths” to aid travelers to reach a certain destination at least at a given arrival time. Gao & Chabini (2006) [34] define a taxonomy of optimal routing problems with the use of policy in the nodes and focus on the variant that assumes a *perfect online information* with all current link travel times available at the moment the algorithm is at one node and has to choose the next node to be reached based on least travel time.

Although considering stochastic events can always enhance any routing system, traffic patterns are usually very consistent over time [27, 89, 23]. This partly justifies the use of functions that attempt to model travel times (or speed of flow) as a function of time of the day—as indeed many traffic information systems do (e.g., [31])—as a sort of “picture of the fluctuation traffic condition” or a priori information source [34]. In this sense, Fleischmann et al. [31] suggest a smoothed step-wise travel time function (Figure 2.3), that is a linearization of the steps at some parametrized point, to overcome shortcomings of jumps of travel time if the raw function generated by the traffic information system is instead used. They also show that not many time slots are required to aggregate collected travel times, as five to ten already provide excellent approximation of the travel times, though the use of more time slots—with slightly improved approximation of travel times—has not significantly influenced the required CPU time to process the raw data. A long history of traffic monitoring should then allow to accurately identify traffic conditions [89] and as such even provide fairly good forecasts, that are of particular interest for planning in vehicle routing.

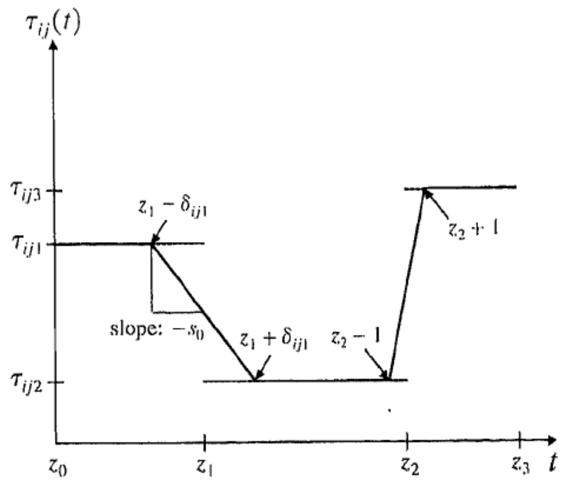


Figure 2.3: Smoothed travel time function (31).

2.3 The role of Shortest Path Problems

If VRP are not only about finding shortest paths, certainly some tasks related to that may safely be reduced to a shortest path computation, whatever the costs associated to the moving from one point to another are. Fleischmann et al. [32] proposed a structure for a dynamic routing system as part of a so-called *observation system* (see Figure 2.2), in which shortest paths calculations play a major role. This system would be responsible for filtering travel time information from the management center, calculating required shortest paths and travel times, as well as to observe changes in travel times for these paths, that eventually trigger other shortest path computations [32]. This brings up the need for systematic evaluation of computed routes, hence requiring shortest path problems to be *dynamic*, or at least to consider the time-varying nature of travel costs, as will be discussed in the following Chapter 3.

Dynamic shortest path problems are not restricted to the context of VRP, but certainly these transportation and logistic problems are clear examples of applications that may benefit from such a setting, as in general any-purpose routing systems would do (e.g., car navigation systems). VRP, however, are much more likely to make use of a setting as the one this research project works with—i.e., shortest path problems to be solved in a server scenario basis.

Chapter 3

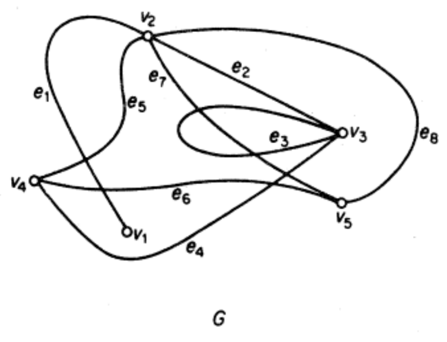
Dynamic Shortest Path approaches

This chapter narrows down to the kind of problems we particularly study in this research, namely *dynamic shortest path problems*, more particularly the ones known as *time-dependent shortest path*. We define what are shortest path problems, as well as present traditional approaches to solve them, and then define what we mean by a *dynamic* shortest path. Solutions for this variation of the problem are the presented, followed by a discussion on speed-up techniques to fast computation of shortest path problems. Finally, we somewhat restrict the scope again by pointing out that focus on dynamic shortest paths in a database context, which is the real objective of this project.

3.1 Dynamic Shortest Path problems

Shortest path problems play a major role in logistics and transportation for general routing planning [24, 26, 80] and also in the context of VRP (see Chapter 2). By the latter we mean that eventually routes need to be defined for the vehicles of a fleet dispatched to attend a set of customers, using a minimum cost route, i.e., a shortest path whatever the costs to be minimized are. Traditionally they are distance or at least distance-dominated functions [14].

As many other engineering and scientific problems [42, 60], determination of shortest path is well-abstracted by means of representing the transportation network as a set of points with lines joining certain pairs of points. This gives rise to the mathematical notation that suits this situation: the concept of a *graph* [16]. Figure 3.1 illustrates the definition of a graph in which $V(G)$ is a nonempty set of vertices (or nodes), $E(G)$ a set of edges (or links) and ψ_G is an incident function that associates with each edge of G a (not ordered nor necessarily distinct) pair of vertices of $G(V, E)$. Those graphs that can be embedded without edge crossings in the plane are called *planar*, which means that edges only intersect in their ends [16]. Translating this to a transportation dialect, that would define a network without overpasses and underpasses, thus usually a transportation network is more likely to be a *non-planar* graph. Moreover, if a graph has no two links that join the same pair of vertices (i.e., no loops), it is a *simple* graph [16]; many transportation networks are abstracted—but this will depend on how the network was modeled—with a single link and the same pairs of vertices to represent an edge that actually allows bidirectional traversal, so they may be *non-simple* .



$$G = (V(G), E(G), \psi_G)$$

where

$$V(G) = \{v_1, v_2, v_3, v_4, v_5\}$$
$$E(G) = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8\}$$

and ψ_G is defined by

$$\psi_G(e_1) = v_1v_2,\ \psi_G(e_2) = v_2v_3,\ \psi_G(e_3) = v_3v_3,\ \psi_G(e_4) = v_3v_4$$
$$\psi_G(e_5) = v_2v_4,\ \psi_G(e_6) = v_4v_5,\ \psi_G(e_7) = v_2v_5,\ \psi_G(e_8) = v_2v_5$$

Figure 3.1: Definition and an example of a graph (adapted from (16).)

Translated to graph theory, the task of finding the shortest path is the one of searching a path that leads from a start node[1] v_s to an target node[2] v_e, both in $V(G)$, while minimizing the costs ψ_G associated to the links in $E(G)$ [42]. If we let there be a real number $w(e)$, that defines the incidence cost ψ_G associated with each edge in the graph, we attempt to find the connected set of edges through this weighted graph that minimizes the cost of traversing from start to target node [16]. This optimal connected set of edges we refer as the *shortest path* ($v_s - v_e$), and with the adoption of the convention that all the weights are positive, a solution can be found using the algorithm[3] of Dijkstra (1959), that actually finds the shortest path from source to all others vertices $V(G)$ [16]. A similar algorithm was developed independently by Whiting & Hillier (1960) [16]. Dijkstra's is indeed "the most commonly used approach for

[1]We will also use the term source node interchangeably.

[2]We will also use the term destination node interchangeably.

[3]A good graph-theoretic algorithm has the computation steps bounded by a polynomial in ν and ϵ, according to Edmonds (1959) [16].

answering shortest path queries (see Figure 3.2), along with variants, with use of some heuristic, that aim at reducing the *search space*" (number of nodes visited by the algorithm) [85]. One of the famous heuristic shortest path algorithm based on Dijkstra is the A* algorithm, which adds the concept of *closeness* [47] to drive the search towards the target node [63]. That leads to a number of visited nodes never larger than in Dijkstra's algorithm, but in norm actually smaller [63].

```
for all nodes $u \in$ V setcost(u) $\leftarrow \infty$
initialize priority queue $Q$ with source $s$ and set cost($s$) $\leftarrow 0$
while $Q \neq 0$
    get node $u$ with smallest tentative cost($u$) in $Q$
    for all neighbor nodes $v$ of $u$
        set new_cost $\leftarrow$ cost($u$) + $\psi(u, v)$
            if new_cost $<$ cost($v$)
                if cost($v$) $= \infty$
                    insert neighbor node $v$ in $Q$ with priority new_cost
                else
                    set priority of neighbor node $v$ in $Q$ to new_cost
                set cost($v$) $\leftarrow$ new_cost
```

Figure 3.2: Dijkstra's algorithm (after (85)).

The most traditional approach for solving shortest path in a transportation context, that is considering distance as the cost associated to the edges (i.e., road segments), is somewhat misleading, particularly when one considers travel time as the cost to be minimized. In the lowest level of this issue, one will point out that different kinds of road (e.g., hierarchy, conditions of traffic, or even actual condition of road infrastructure) will allow diverse flow characteristics, eventually leading to travel times that do not reflect the logical rule: shorter segment, shorter travel time. What's more, travel times may be time-dependent, due to conditions of traffic [31, 32, 43, 44, 70, 46, 84, 57, 49] or more generically stating, following certain *environmental variables* [73]. It might also happen that unpredictable changes occur in the network data, leading to a reoptimization problem,[4], which together with time-dependency brings up the concept of *dynamic shortest paths* [21]. Essentially, they are nothing but shortest path versions that need to be performed in a *dynamic graph* [64]. Nannicini & Liberti (2008) [64] define a dynamic graph as one in which any of the components (i.e. edges, vertices or costs) change as function of time. Most commonly it is the cost that change, following conditions of traffic. In such a setting, usually we talk about travel time, directly or indirectly, as the cost to traverse an edge in the graph, and that varies as a function of the time of the day. As we all know that you better off cutting through some auxiliary roads rather than crossing that main road at peak time, for instance.

[4]To reoptimize a shortest path tree is usually faster than recalculating it from scratch [64] therefore it may be worthwhile to monitor changes in travel times and reoptimize the shortest paths on demand.

TDSP problems have many applications [63, 64], including transportation and logistics [8] related optimization tasks, and their conceptualization dates back to 1966, "when Cooke & Halsey proposed it in a discrete time manner" [21]. In such problems, the travel time to traverse an edge of the network is assumed to be predictable or previously known as a function of departure time [21], though some uncertainty may be present [86]. The general TDSP problem is at least NP-hard [3, 20, 21], i.e. they are as hard as any problem in NP (non-deterministic in polynomial time) [33], and theoretically an algorithm for solving it could be translated to solve any other NP-problem [18]. But this notion of hardness will be always related to the characteristics and size of the problem at hand. In this sense, Impagliazzo [48] states that "there is a gap between a problem being not easy to being difficult, meaning that many problems, though not solvable in its worst-case algorithm, may still be solvable for *most* instances, or in instances that arise in practical situations". Moreover, depending on how one defines the problem, it may even well not be in NP, as is TDSP under the assumption of FIFO[5], also called a *non-overtaking property* [63] and the properties that arise from it, which enables the development of efficient polynomial-time solutions [20, 21, 1, 62, 64, 63]. Nonetheless, still the performance will also be influenced by how complex the time-dependent network data functions are (in other words, how much travel times vary along the network), and very often "solutions will consist of many linear pieces that lead to slower computation and greater memory demand" [20]. This in many cases justifies the use of approximate solutions, i.e. *discrete-time solutions* (see Section 3.2.1), considering that delivering exact solutions in *continuous-time* would be in some cases only possible by means of a parallel computing environment [20, 75]. To overcome some of the slow behavior of continuous-time solutions, Dean (1999) [20] theoretically presents the idea of *hybrid continuous-discrete algorithms* that assume behavior of a discrete-time algorithm when a fine time scale is identified in the solution dynamics.

TDSP problems can be reduced to two fundamental problem variants: earliest arrival (EA) or latest departure (LD) problems [21]. What differentiates one from the other is only whether the constraint in time is associated to the destination (LD) or source node (EA). Dean [21] specified a notation for the different *flavors* of the two fundamental TDSP problem variants, as well as briefly presents how one could solve them (Table 3.1). The different *flavors* or versions of the two fundamental problems only define whether one wants to solve for specific (s, d, (t)) or all possible (*) nodes (spatial constraint) and/or time (temporal constraint).

Bérubé et al. [15] show a classification of TDSP problems which is an adaptation of characteristics in [17] (Table 3.2). In a sense, they extend the fundamental problems variant presented in Table 3.1 by considering that other issues like objective, whether waiting in nodes is allowed as well as network properties help to properly formulate a TDSP problem.

In practice, different approaches for solving time-dependent shortest path

[5]FIFO, fist-in-first-out concept, is essentially an assumption that may or may not be taken when dealing with a network that defines whether is possible to traverse an edge later and still reach its end earlier.

Table 3.1: Reduction to two fundamental problem variants and methods of computation [21].

Desired Output	**Method of Computation**
$EA_{s*}(t)$, $EA_{s*}(*)$	These are the two fundamental problems, from which we can express all other variants.
$EA_{sd}(t)$, $EA_{sd}(*)$	As with the static shortest path problem, the single-source single-destination problem seems just as difficult as problems involving either multiple sources or multiple destinations. We therefore solve these as the more general problems $EA_{s*}(t)$ and $EA_{s*}(*)$, respectively.
$EA_{**}(t)$, $EA_{**}(*)$	These are solved by performing n computations of $EA_{s*}(t)$ and $EA_{s*}(*)$, respectively, for every s ∈N.
$EA_{*d}(t)$	One can show that this problem is no easier than computing $EA_{**}(t)$.
$LD_{sd}(*)$, $LD_{s*}(*)$, $LD_{*d}(*)$, $LD_{**}(*)$	These are computed by solving for and inverting the corresponding earliest arrival time functions. For example, $LD_{s*}(*)$ is found by solving for $EA_{s*}(*)$ and taking inverses.
$LD_{s*}(t)$	One can show that this problem is no easier than computing $LD_{**}(t)$.
$LD_{*d}(t)$	Performing a time-reversal transformation on the network transforms this into the problem of computing $EA_{s*}(t)$.
$LD_{sd(}t)$	Solve the more general problem $LD_{*d}(t)$.
$LD_{**}(t)$	Solve $LD_{*d}(t)$ repeatedly for every d ∈N.
$EA_{*d}(t)$	Performing a time-reversal transformation on the network transforms this into the problem of computing $LD_{s*}(*)$.

problems have been proposed in the literature. Ding et al. [28] discuss three types of algorithms, namely *discrete-time*, *Bellman-Ford based* and *A** (equivalent to Dijkstra) algorithms. We will discuss each of these types of algorithms, plus the one proposed by Ding et al. [28] in the next section.

3.2 Dynamic Shortest Path solutions

3.2.1 The Discrete-time algorithm

An efficient discrete-time algorithm for dynamic networks was presented by Chabini [17]. At that time, he claimed to have developed “the most efficient algorithm to what appeared to be a 30 years old problem” of computing all-to-one shortest paths in discrete dynamic network. He also pointed out its application at the heart of efficient solutions approaches to network models in dynamic transportation systems, such as Intelligent Transportation Systems applications. Among other things, Chabini [17] defined shortest path problems as dynamic or time-dependent (interchangeably), and distinguished different

Table 3.2: Classification of TDSP problems after (15).

Criteria	**Variant**
Objective	*Minimum cost path* (general formulation);
	Fastest path: minimum cost path with cost = delay;
	Bi-criteria path: optimization with two criteria (e.g., cost and time);
	Multi-criteria path: optimization with more than two criteria;
Waiting constraints	*Forbidden waiting*: waiting at a node is forbidden;
	Unbounded waiting: waiting is allowed and is unbounded;
	Bounded waiting: waiting is allowed and is bounded;
Waiting costs	*Memoryless*: waiting costs are independent of waiting duration;
	With memory: waiting costs depend on waiting duration;
Source and destination	*One-to-all*: compute paths from one source to all destinations;
	All-to-one: compute paths from all sources to one destination;
	One-to-one: compute paths from a single source to a single destination;
Network properties	*FIFO network*: FIFO property is satisfied;
	Periodic network: cost and delay follow periodic patterns;
	Zero-delay arcs: arcs with zero-time delays are allowed;
	Travel beyond time horizon T.

dynamic shortest path problems depending on several characteristics as displayed in Table 3.2. Among them, the network FIFO property that allows polynomial computation time [20, 21, 1, 62, 64, 63]. Many transportation networks exhibit FIFO behavior [21], and as a rule, inland transportation networks can be generally assumed to be of that class. Therefore it does not make sense to allow to wait at a certain node unless it is for another reason than taking a travel time advantage [1].

Restricted to a discrete-time representation, Chabini's [17] algorithm find approximately a *least total travel time* (LTT) by exploding the graph into k time points evenly, allowing then the TDSP problem to be solved as a static single source shortest path problem in the exploded graph. Two fundamental drawbacks are inherent to discrete-time approaches [28]:

1. Difference between approximated LTT obtained and the optimal LTT, called LTT error (accumulative way along the paths), which is very sensitive to the parameter k in which the time dimension is discretized.

2. Whilst increasing k may lead to a closer to optimal result, the exploded graph is k times larger than the original.

3.2.2 The Bellman-Ford based algorithm

Orda & Rom (1990) [66] addressed time-dependent shortest path problems without the restriction of a discrete time representation and positive integers as domain and range. Their—which we refer to as OR90—algorithm (Figure 3.3) is a generalization of the Bellman-Ford shortest path algorithm [28] and takes a time-dependent graph $G_T(V,E,W)$ with nodes V, edges E and costs W.

In the OR90 algorithm, $g_l(t)$ are the *earliest arrival time* at node v_l from start node v_s, while $h_{k,l}(t)$ are the *earliest arrival time* at v_l from start node v_s traversing edge (v_k, v_l), both for starting time $t \in T$. These two functions are updated until convergence to the correct values, and after computing the best starting time t^*, the optimal path p^* can be constructed based on $g_l(t)$ and $h_{k,l}(t)$ functions (see [66]).

```
Algorithm 1 OR90($G_T(V, E, W), vs, ve, T$)
for all $v_l$ ∈ V do $g_l(t)$, ← ∞ for t ∈ T;
for all ($v_k$,$v_l$) ∈ E do $h_{k,l}(t)$ ← ∞ for t ∈ T;
$g_s(t)$ ← t for t ∈ T;
repeat
    for all ($v_k$,$v_l$) ∈ E do $h_{k,l}(t) \leftarrow g_k(t) + w_{k,l}(g_k(t))$;
    for all $v_l$ ∈ V do $g_l(t) \leftarrow min_{v_{k \in N(vl)}} \{h_{k,l}(t)\}$;
until all functions $g_l(t)$ remain unchanged
return ($t^* \leftarrow argmin_{t \in T}$ {$g_e(t)$ - $t$}, $p^*$);
```

Figure 3.3: Orda Rom algorithm ((66) after Ding et al. (28))

The algorithm takes a strategy of determining paths toward destination v_e while refining the arrival-time functions $g_i(t)$ for the whole interval T, which can be called a *path selection and time refinement* approach [28]. Though it has the characteristics of treating time as continuous, costs not only as integers and positives, as well as handles arbitrary functions for link delays [66], the high time complexity ($O(nm\alpha(T))$, where $\alpha(T)$ is the time required in a function operation in interval T, n = $|V|$ and m = $|E|$, makes it unfeasible to apply it on large and dense time-dependent graphs [28]. Combining path selection and time refinement, "the algorithm does not recognize when a function is already well-defined for a subinterval of T, thus it always need to recalculate for the whole interval T and the convergence is therefore slow" [28].

3.2.3 The adapted A* algorithm

Kanoulas et al. [50] provided an extension to the A* algorithm to handle TDSP problems, which is also a *path selection and time refinement* approach [28]. But again the path selection and time refinement are coupled—even more than in the algorithm of Orda & Rom [66]—leading to a worse case in which all $v_s - v_e$ paths are enumerated and the time/space complexity is exponential with the size of the graph G_T [28]. Some practical findings about performance on Kanoulas et al. [50] approach are as follows [28]:

- it can only perform efficiently when estimation can assist pruning the search space effectively and v_s and v_e are close to each other in graph G_T;
- as estimation to prune search space is difficult in general graphs, it is infeasible to handle large time-dependent graphs, where v_e may be far away from v_s.

Given the fact that A* heuristics algorithm usually provide better results in static shortest path solution, there are more attempts of adapting that algorithm for time-dependent shortest paths. We will refer to more works in this direction: Huang et al. [47], Delling & Nannicini [25], Zhao & Ohshima [91] and Delling [24] for a good overview on route planing including A* search heuristics adaptations.

3.2.4 The Two-step time-dependent shortest path algorithm

Ding et al. (2008) [28] also recognized the concept of *time refinement and path selection* as promising for solving time-dependent shortest path problems, particularly when an interval of departure time is given for optimizing. It follows that the main characteristic of their algorithm is the decoupling of time refinement and path selection, giving rise to a two-step least travel-time algorithm (named Two-step LTT by the authors). The outline of their algorithm can be found in Figure 3.4, and generally treats the TDSP problem under the FIFO assumption, but the authors also show that it is possible to apply it for non-FIFO networks (see Section 4.6.1). It takes a time-dependent graph G_T and a query LTT(v_s, v_e, T) with a start node v_s, and a target node v_e, and a departure time interval $T = [t_s, t_e]$ as input, returning an optimal $v_s - v_e$ path $p*$ for an optimal departure time $t*$. They make use of edge-delay functions of class *continuous piecewise-linear*, thus the problems of discontinuities in the functions that make Dijkstra-based algorithms to fail in finding TDSP [66] do not apply in this case. For dealing with functions with discontinuities,[6] we refer to the works like Dell'Amico & Pretolani [22].

The first step of Ding et al. (2008) [28] algorithm determines *least arrival-time functions* $g_i(t)$ for each node v_i of the graph in the give interval time T (Figure 3.5), performing as a Dijkstra algorithm. The main advantage of this strategy of decoupling time refinement and path selection is the ability of identifying a subinterval for which the arrival-time function is already well-defined, therefore it does not need to be computed again. The lack of this idea is what makes the Orda & Rom (1990) [66] algorithm slow in converging to the arrival-time functions [28].

[6]This is an issue that may apply in real-world applications, as for instance when travel-time functions are dynamically generated and straightaway put to use in a routing system (e.g., [32].

```
Algorithm 2a Two-Step-LTT$(G_T(V,E,W),vs,ve,T)$
$\{g_i(t)\} \leftarrow timeRefinement(G_T,v_s,v_e,T)$;
if $\neg$ (g$_e(t)=\infty$ for the entire $[t_s,t_e]$ then
    $t^* \leftarrow argmin_{t\in T}\{g_e(t)-t\}$;
    $p^* \leftarrow pathSelection(G_T,\{g_i(t)\}vs,ve,t^*)$;
    return $(t^*,p^*)$;
else return $\emptyset$;
```

Figure 3.4: Outline of Two-step least travel-time algorithm (28) .

The second and last step of the algorithm is to perform a fast path selection. This is done after the optimal departure time followed the refinement of arrival-time functions, which is defined simply as the departure time in which the arrival-time minus the departure time from the start node (i.e., $g_e(t)-t$), in the arrival-time function of the target node is minimal, or formally $argmin_{t\in T}\{g_e(t)-t\}$. Path selection occurs in a backward manner, allowing to identify, from target node v_e, step-by-step, the predecessor nodes until it reaches the start node v_s (Figure 3.6). We refer to Section 4.3.1 to further explanations on the approach.

Ding et al. [28] conducted a series of experiments, first varying the number of nodes, the density of the graph (i.e., increase of edges for a fixed number of nodes) for a fixed departure time interval $T=[0,500]$, and then also varying the departure time interval from 50 up to 1,000. A varying distance between source v_s and destination v_e was also tested. In general, they demonstrate that their algorithm outperforms the three others (discrete-time algorithm [17], Bellman-Ford adapted algorithm [66] and A* extended algorithm [50]) in both time and space complexity. There are a few cases in which another algorithm, namely A* extended [50], performs better than the Two-step LTT, which is when number of nodes or edges is small, or the distance between v_s and v_e is short. But that only reinforces the argumentation of Ding et al. [28] on the limitations of Kanoulas et al. [50] in dealing with large graphs, as the performance of their algorithm drastically reduces when the conditions of small graph or short distance are not met.

3.3 Speed-up techniques for shortest path computation

Shortest paths over large-sized (i.e., real-world) graphs is an expensive computation task even in its static sense. In a transportation perspective, for instance, several studies state that Dijkstra's algorithm is a prohibitive choice for large road networks [76, 80, 27], though largely solve the problem from a worst case perspective [76]. As a matter of fact, better performing algorithms are usually reported as in comparison to Dijkstra's performance over the same request set (e.g., [76, 80, 26]). To be fair with Dijkstra's algorithm, one should always noticed that any eventual faster algorithm for computing shortest path

```
**Algorithm 2b** $timeRefinement(G_T(V,E,W), v_s, v_e, T)$
$g_s(t) \leftarrow t$ for $t \in T; \tau_s \leftarrow t_s$;
for each $v_i \neq v_s$ do
    $g_i(t) \leftarrow \infty$ for $t \in T; \tau_i \leftarrow t_s$;
Let $Q$ be a priority queue initially containing pairs, $(\tau_i, g_i(t))$,
for all nodes v$_i$v$_i$ $\in V$, ordered by $g_i(\tau_i)$ in ascending order;
while $|Q| \geq 2$ do
    $(\tau_i, g_i(t)) \leftarrow dequeue(Q)$;
    $(\tau_k, g_k(t)) \leftarrow head(Q)$;
    $\Delta \leftarrow min\ \{w_{f,i}(g_k(\tau_k))\ |(v_f, v_i) \in E\}$ ;
    $\tau_i' \leftarrow max\{t \mid g_i(t) \geq g_k(\tau_k) + \Delta\}$;
    for each $(v_i, v_j) \in E$ do
        $g_j'(t) \leftarrow g_i(t) + w_{i,j}(g_i(\tau_i)$ for $t \in \left[\tau_i, \tau_i'\right]$;
        $g_j(t) \leftarrow min\{g_j(t), g_j'(t)\}$ for $\in \left[\tau_i, \tau_i'\right]$;
        $update\,(Q, (\tau_j, g_j(t)))$;
    $\tau_i \leftarrow \tau_i'$;
    if $\tau_i \geq t_e$ then
        if
            return $\{g_i(t)\,|v_i \in V\}$;
    else
        $enqueue\,(Q, (\tau_i, g_i(t)))$;
return $\{g_i(t)\,|v_i \in V\}$;
```

Figure 3.5: Ding et al. (2008) (28) time refinement algorithm.

come either at the expense of a perhaps non-optimal solutions—like in many implementations on commercial routing systems [76, 77, 10]—or by a speed-up technique that demands data preprocessing. Dijkstra's algorithm performance is also dependent on implementation [30] and fairly good improvements in actual running time can be achieved by implementing Dijksta's algorithm more efficiently (see [90]). In reality, what many speed-up techniques do is to find a way of stopping Dijkstra's search once one is guaranteed that the shortest path has been found [76]. According to Delling et al. [26], research on speed-up techniques for solving shortest paths started in the late nineties, but a real "horse race" for getting the fastest query and preprocessing times began when continental-sized networks became available for research in 2005.[7] We refer to their work for a chronological summary of reported speed-up techniques as compared to the classical Dijkstra's algorithm; but also point out that improvements of several million times are already a reality in the context of static shortest path computation [26].

An usual technique for speeding-up the computation of shortest paths is to search simultaneously forward and backward [77, 60]. This approach is called *bidirectional search* and proof can be derived that once some node was visited

[7]By the company PTV AG.

```
Algorithm 2c pathSelection($G_T(V,E,W)$,$\{g_i(t)\}v_s, v_e, t^*$)
$v_j \leftarrow v_e$;
$p^* \leftarrow \emptyset$;
while $v_j \neq v_s$ do
    for each $(v_i, v_j) \in E$ do
        if $(g_i(t^*) + w_{i,j}(g_i(t^*)) = g_j(t^*)$ then
            $v_j \leftarrow$
    $p^* \leftarrow (v_i, v_j) \cdot p^*$;
return p*;
```

Figure 3.6: Ding et al. (2008) (28) path selection algorithm.

from both directions, the shortest path can be found from what has been already computed (refer to [30]). The searches are executed at the same moment over a *forward graph* $\overrightarrow{G}(V, \overrightarrow{E})$ and a *backward graph* $\overleftarrow{G}(V, \overleftarrow{E})$ [77], starting from the start node v_s and the target node v_e respectively. Terminating condition is that the search frontiers have met [77, 80, 26, 60] (see Figure 3.9b), or formally stating that there is node v_i to/from which a shortest path has been found both from v_s and to v_e.[8] This technique applied to Dijkstra's algorithm can be expected to provide a speed-up of factor two , i.e., an exploration of half of the number of nodes, when compared to the traditional unidirectional search [60]. What's more, bidirectional search is a flexible technique that can be combined with most of other speed-up techniques [26, 60], and in fact is an component of many advanced ones found in the literature (e.g., [39, 76, 80, 60]). It is also a speed-up technique that does not demand any kind of preprocessing step, hence it is out of the trade-off between precomputation time, need for storage of precomputed data and query time that should be considered when devising speed-up methods [76, 26].

Apart from bidirectional search, Sanders & Schultes [77] distinguishes two basic speed-up approaches: (1) the search is directed towards the target node (and towards the source node when there is a backward search); (2) the search exploits inherent *hierarchy* levels, particularly in road networks. In some cases, a mixture of the two is eventually reached by storing more information [77]. We discuss some techniques of the two approaches under the denomination of "goal-oriented" and "hierarchy-based" routing in the following. An overview of shortest path algorithms and speed-up techniques is shown in Figure 3.7.

3.3.1 Goal-oriented routing

The A* algorithm. It differs from Dijkstra's algorithm for it takes into account the concept of "closeness" to the target node as a heuristic to drive the search towards its destination [47]. In this case, when priority keys are defined for the queue used in the algorithm's search, an additional potential func-

[8]Which not necessarily mean that v_i will be part of the shortest path, but one can be assured that all the necessary nodes to found the shortest path has been visited once that happened [30].

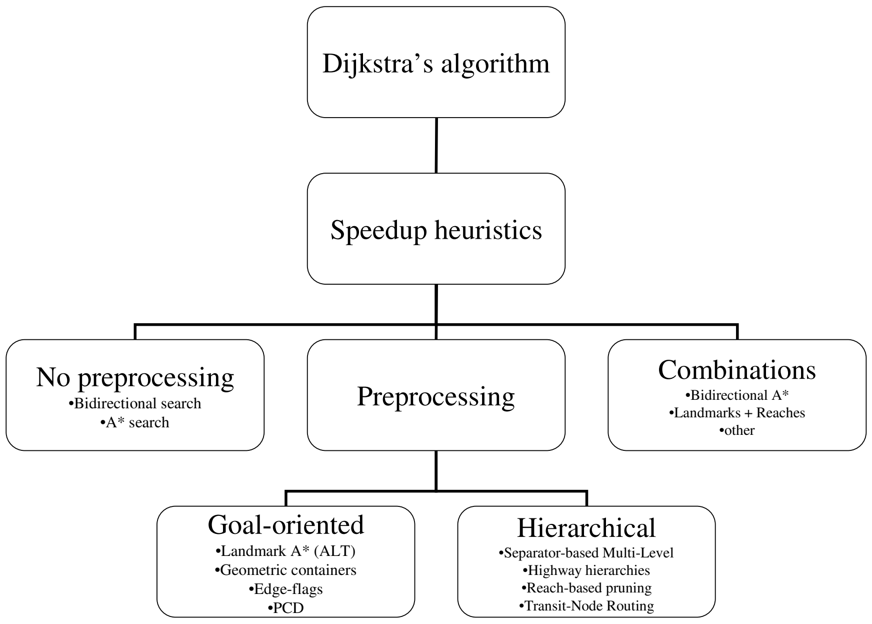


Figure 3.7: An overview of shortest path algorithms and speed-up heuristics (adapted from (60)).

tion is defined to estimate the cost[9] between any node v_i and the target v_e with the effect of giving priority to nodes that are supposed to be closer to the target [63]. There is, though, a condition to be satisfied for A* to always provide optimal paths, which is that the estimation of the cost of node v_i to the target should never overestimate the real cost between them, or formally $\pi(v_i) \leq c(v_i, v_e) \forall v_i \in V$ [63]. When the cost considered is distance, one usual heuristic for underestimating the costs in π is by computing the Euclidean distance between start and target node [47, 26]. Nodes are considered better candidates to continue the search based on the estimation π. A* algorithm actually defines the priority function by balancing two costs, the traditional cost $c(v_i, v_j)$ to reach a next node connected by an outgoing edge from the visited node in iteration and the estimation in π (i.e., $f(v_i) = c(v_i, v_j) + \pi(v_i)$). The speed-up comes from the reduction in the search space (see Figure 3.9c) which guarantees that the number of visited nodes is never higher than in the Dijsktra's algorithm [63]. How much gain can be obtained depends on the heuristics used to define the potential function, and in its basic implementation (i.e., Euclidean distance estimation) still a significant part of the visited nodes will never be a member of the shortest path [47]. A drawback of using A* with heuristic of Euclidean distance between start and target node may be the need for having the layout of the graph [60] (i.e., its actual form on Euclidean space) stored along with the graph's representation. On the other hand, it is a reduction of search

[9]In these descriptions we use the generic term *cost* but in most of the cases this cost is distance.

space method that can eventually be used together with other speed-up technique for even faster results (e.g., [39]; see Section 3.3.2), and that has been show to be adaptable to situations of dynamic networks [47, 25, 91].

A* search with landmarks—The ALT algorithm. A layout-independent way of using A* algorithm is to use the concept of *landmarks* as heuristic to compute the potential function [63]. It is based on the selection of nodes $L \subset V$, called landmarks, to which the costs to/from any other node $v \in V$ is pre-computed (or more formally, $c(v,l), c(l,v) \forall v \in V, l \in L$) [63]. Strong bounds to shortest path costs can be obtained by using this approach, even with only the use of ≈ 16 landmarks well-spread along the extremes of the network [37]. This approach provides a lower bound of the cost $c(u,v)$ following the triangle inequalities $c(l_1,u) + c(u,v) \geq c(l_1,v)$ and $c(u,v) + c(v,l_2) \geq c(u,l_2)$; thus $\pi(v_i) = max_{l \in L}\{c(u,l) - c(v,l), c(l,v) - c(l,u)\}$ [63, 24]. An illustration can be seen in Figure 3.8. As lower bounds and consequently the speed-up of computations using ALT are dependent on the chosen set of landmarks to better estimate the actual cost $c(u,v)$, some heuristics have been proposed in literature to perform this task (see [24]). Goldberg et al. [38] report a reduction in search space of factor 44, but that only leads to a reduction of factor 21 in query time, when compared to Dijsktra's algorithm.

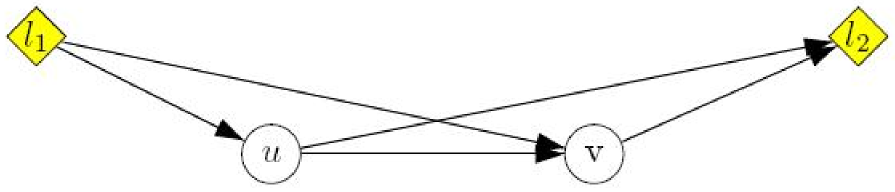


Figure 3.8: Landmarks (l_1, l_2) and triangle inequalities intuition [24].

Geometric containers. It is somewhat related to the classic A* search space pruning, as the main idea is to define a constrained space to restrict the nodes in the search to those that are more likely to be part of a shortest path $v_s - v_e$. Wagner et al. [85] investigate twelve possible geometric containers to prune the search space, which are created in a preprocessing phase of linear computational complexity to the size of the input graph. A Dijkstra's algorithm can be applied over the pruned search space, or a combination with A* algorithm can be devised in a straightforward manner. We refer to their work [85] for details on the investigated types of geometric containers. For the surprise of the authors, the simple "bounding box" outperformed other geometric containers in many cases; also, the speed-up factor seems to not have relation to the size of the graph, therefore the method is scalable. An adaptation for a dynamic situation in which the edge cost changes is given through updates in the containers 2-3 faster than to compute new geometric containers from scratch, while enabling to maintain a pruned search that leads to optimality in almost all cases. A limiting factor for this approach is the "large-graph prohibitive requirement for computing one single-source shortest path from every node while preprocessing the geometric containers" [60]. A similar technique is the so-called *edge-flags* in which a precomputed flag indicates whether an edge is contained in a shortest path to any node in a given partition [60].

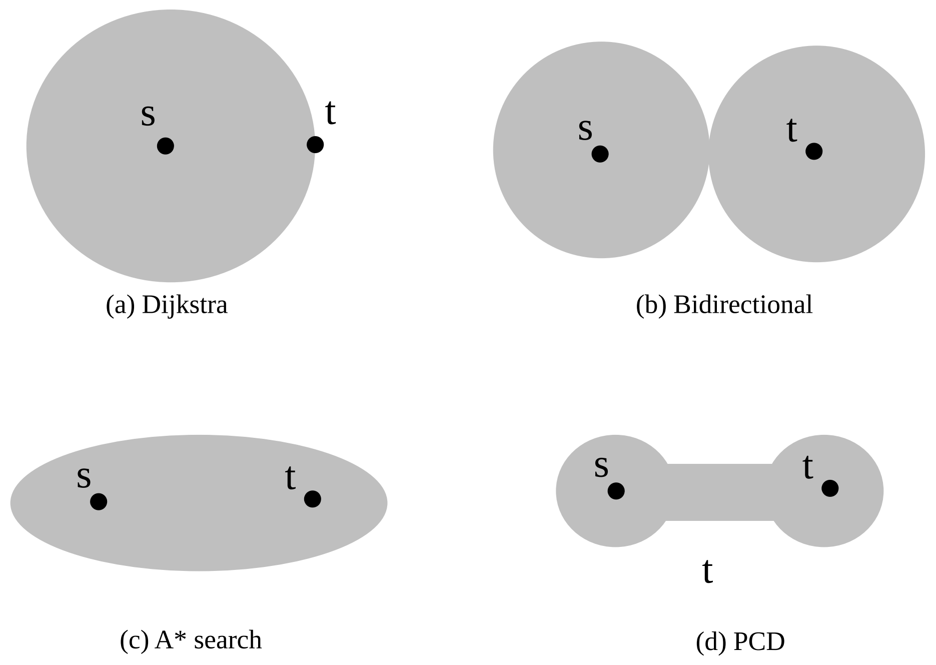
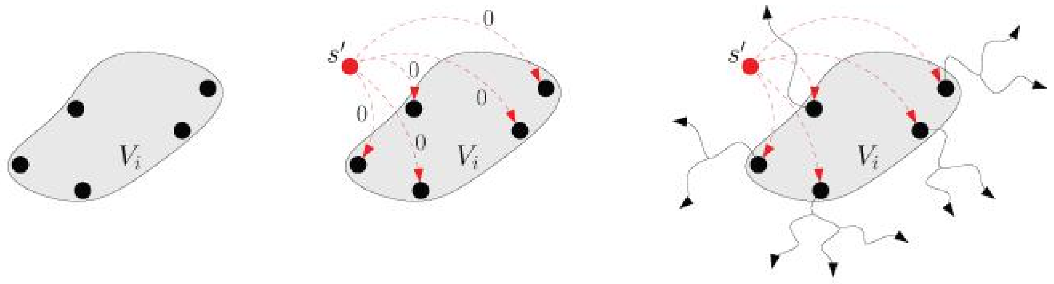


Figure 3.9: Idealized shape of the search space for shortest path computations in different methods (after [60]).

Precomputed Cluster Distances—PCD algorithm. It was presented in Maue et al. [60] and consists of the precomputation of minimum costs for any combination of nodes, taking each node from a cluster. The graph is partitionated in k clusters somehow, i.e., using any kind of partitioning method. Performance is dependent on the partition mechanism and the preprocessing stores minimum costs for all k^2 pairs of clusters. Some advantages of PCD over related methods are: number of clusters k can be adjusted to decrease preprocessing time; independent on the partitioning method (i.e., number of border nodes) since exactly k single-source shortest path computations are performed; even simple grid clustering method can achieve high speed-up; and it does not necessarily need the layout of the graph to perform the partition, though this might depend on the chosen clustering method. Preprocessing is restricted to only k single-source shortest path computations by adding a "dummy node" s' connected to all borders of a given cluster S with a zero-cost-edge (see Figure 3.10 for an illustration). Different methods of partitioning were tested by Maue et al. [60], demonstrating that a strong sense of target direction is achieved by the method, which leads to speed-up of about 115 (compared to Dijsktra's algorithm) in terms of average query time. The authors conclude that a good partitions are equal size and low diameter clusters, and suggest a combination with hierarchical techniques to either provide even better speed-up or to use for quickly precomputation of cluster distances.



(a) Cluster V_i with its border nodes. (b) Dummy node s' and zero-weight edges. (c) Shortest-paths search from s'.

Figure 3.10: Preprocessing connections from a cluster V_i (60).

3.3.2 Hierarchy-based routing

Exploitation of hierarchy levels in road networks is what makes commercial routing systems able of handling large graphs, but that in many cases come at cost of not guaranteeing optimal routes [39, 76, 10]. They usually rely on a simple intuition that once one is going "far enough", one will only consider to traverse a few access routes until a network that is relevant for long-distance travel is reached; from this point on, one may only consider this rather sparse network until eventually be "close enough" of another set of few access routes to reach the destination. This is far from a wrong observation [39, 10], but the problem is that the heuristics that are normally used to perform such a hierarchical approach in commercial systems rarely ensure that optimal routes are always computed. That is because they usually define which roads are important for long-distance travel only based in attribute levels [39, 76, 10].

Academically, then, the point is how to exploit this inherent hierarchy of road networks in such a way that always result in optimal shortest paths. Several approaches have been proposed in literature to do so (see Figure 3.7); we discuss some of them in the following.

Reach-based routing. It was introduced by Gutman [39] as a scientific way of discovering which roads are important for long-distance travel as well as to be flexible for changes in importance (e.g., bad traffic conditions). The principle of reach of a node is the intuition that if it lies on a shortest path that longer extends in both directions, more important is the node. Some claimed advantages of the method are: guaranteed optimality; computation time comparable to commercial approaches (which not always give optimal routes), and looking linear under some conditions; easily combinable with other techniques (e.g., A* search); precomputed data does not require much storing; preprocessing may be fast enough to allow dynamic changes in the network costs for some applications; and it allows faster computation of multiple origins or destinations shortest paths. Reach-values computation can be performed exactly, at cost of more precomputation time (an average of about 16 times greater, but rising up to about 26 times in the larger graph tested), or using a estimated bound. Exact reach provides the fastest computational results for shortest paths (around 15 seconds for an average of 56 km of route), but a combination of estimated bound reach with A* search performs similarly in computing shortest paths (around 17 seconds for an average of 56 km of route) with much less precomputation. Delling et al. [26], however, state that the precomputation time was still prohibitive for large networks; indeed, Gutman [39] tests over networks at most with 393,368 nodes, while the largely used continental-sized networks made available for scientific use in 2005 have above 18 million nodes each, covering Western Europe and USA/Canada [77, 80, 10, 26, 60]. According to Delling et al. [26], the 2006's improvement of reach-based routing turned out precomputation faster and more accurate by adapting Highway Hierarchies preprocessing techniques; still the preprocessing time and query time are slower than Highway Hierarchies, and only the latter can be comparable when a goal-oriented search is combined with reach-based.

Highway Hierarchies—HHs approach. The idea of automatically computing highway hierarchies was proposed in Sanders & Schultes [76] and then enhanced in [77]. Exact shortest paths are delivered by a proper way of defining *local search* and *highway network*. Local search is performed visiting the H[10] closest nodes from start or target node, and edges that do not connect a node within the H closest nodes from the source s to another within the H closest nodes from the target t are called highway edges (Figure 3.11). Formally, let $SP\langle s, \ldots, u, v, \ldots, t\rangle$ be the shortest path between source and target, then $\forall(u, v) \in E \Rightarrow (u, v) \in SP \wedge u \notin H_s \wedge v \notin H_t$. Further speed-up is achieved by collapsing highway edges along a path consisting of node of degree two (first

[10] H is a tuning parameter.

implementation is on undirected graphs) into a single edge [76]. The approach iteratively arrives at multi-levels of highway network to which a bidirectional Dijkstra-like search is performed over a single graph with all the levels [76]. The second version of HHs [77] reduced query time from about 7 seconds in the first implementation [76] to only around 1 second, but also preprocessing was significantly (at least 60%) reduced. They actually provide two different settings for the trade-off between the precomputation time, the storing requirement and the query time: the default HHs almost doubles the storing requirement, precomputes about 90% faster and returns queries in less than 1 second, while the HHs with reduced memory requires around 20% less storing, precomputes about 60% faster and returns queries in over 1 second. Delling et al.[26] recognized the HHs approach as the first one to provide millisecond query times in large road networks, with reasonable amount of preprocessing time (15-55 minutes [77]).

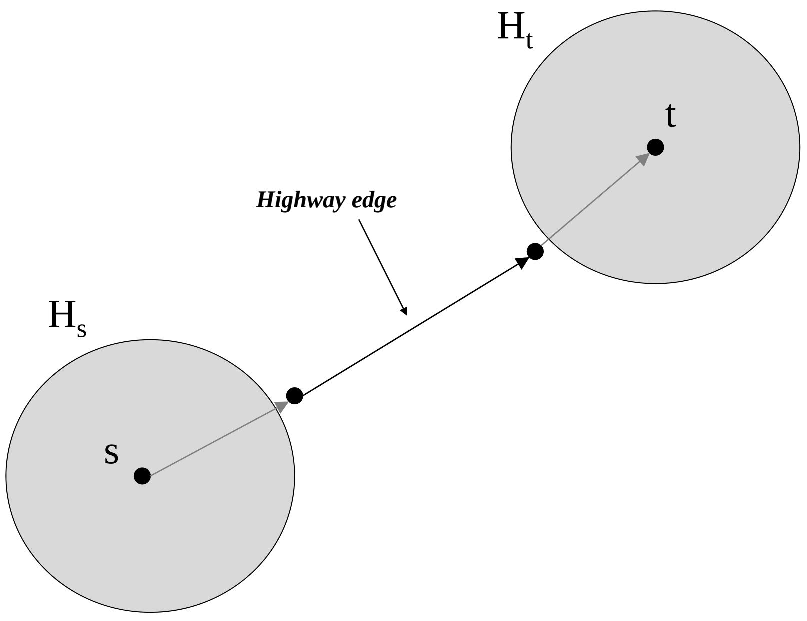


Figure 3.11: Intuition of identifying a highway edge.

Transit-Node Routing—TNR approach. Impressive query time results using transit-node routing are presented in Bast et al. [10], after the introduction of the concept in [9]: between 5 (for global queries) and 20 microseconds (for local queries). As other hierarchical based routing approaches, TNR starts from the intuitive observation that commercial routing systems use to provide much faster (though not always optimal) routes computation: the fact that for long-distance travels one will only consider a rather sparse network of higher-importance roads. This high hierarchy network is formed by a relatively small set of "transit nodes", to which all pairs combination are precomputed [10]. The selection of transit nodes can be performed in different ways (see [26]), and a locality filter ensures that always optimal paths are obtained by effectively defining what is "far enough" to consider the precomputed information [10]. Additional layers of transit nodes are used to 'close the gap' of the locality filter [10, 26] that allows 98.7% of all queries to be treated using a few lookups (refer to [79]). Precomputation time is less than 3 hours and precomputed information consumes 4.5 gigabytes for the network of Western Europe [10]. Therefore, the impressively reduced query time (the best on road networks known until 2007 [80]) of TNR comes at a cost of larger preprocessing and additional space requirement for the precomputed information [26].

3.3.3 Combination of speed-up techniques

In many cases it is possible to combine different techniques to eventually achieve faster computations of shortest paths, or a better trade-off between preprocessing time, storing need for precomputed information and query times. In general, the simple speed-up technique of bidirectional search is recognized as compliable to most of other techniques [26, 60]. Maue et al. [60] point that the combination of bidirectional search with the pruning of a A* can also be devised to an eventual synergy of speed-up as compared to only applying one of them. They also suggest their method, PCD, as a good replacement for landmark A* (ALT) due to its more flexible trade-off between preprocessing time, need for storage of precomputed information and query time. Nonetheless, ALT algorithm has been reported to harmonize well with reach-based routing [38], as well as the latter with the simple A* search as initially tested when reach-based routing was proposed [39]. Wagner et al. [85] indicate that geometric containers could be easily combined with A* search, and also perhaps with bidirectional search, but the latter would come at cost of double preprocessing space and time. The fastest query times known for road networks are believed[11] to be the combination of edge-flags with transit-node routing [26].

3.3.4 Techniques for a dynamic setting

As static shortest path can be considered to be largely solved even for considerably dense and huge road networks like Western Europe and USA/Canada, at fast query times and reasonable preprocessing and additional storage [27, 23], attention of research gradually started to move towards a more dynamic setting. That was also, of course, due to recent technological developments that allow to generate and include up-to-date information on traffic condition as discussed in Chapter 2. At any case, it is clear that dynamic routing is a topic that has recently started to earn more attention, once computation of optimal

[11]This information may be uncertain already.

routes in transportation is one of the "showpieces of real-world applications of algorithmics" [77, 26].

Lifelong Planning A*—LPA* algorithm for moving object. Huang et al. proposed an adaptation of the LPA* algorithm that is able to cope with dynamics in the network while the object (e.g., vehicle) is en route [47]. LPA* is an incremental version of A* that reuses results of previous searches to speed up the subsequent ones (see [52]). Besides the LPA* for dynamic shortest paths, Huang et al. [47] also have developed a new search heuristic as well as a constraint to the space search by applying a minimum bounded rectangle. The adaptation for dynamic network is devised by switching around start and target node when computing the path, while the new search heuristic comes from the intuition that the shortest path is likely to be as close as possible of a straight-line. Thus, to the classic A* search heuristic (Euclidean distance) a further refinement is given by the distance of the path to that straight-line between start and target node, resulting in a priority given to search nodes that are as close as possible to the idealized straight-line path. Finally, the minimum bounded rectangle constrained search is delivered by using the network distance as the principal axis of an ellipse with start and target nodes being the foci; the bounding box of this ellipse forms the constrain for the search space. The algorithm does not always provide optimal solution, but the accuracy in the experimental tests reached 99%. Moreover, the number of examined nodes can be reduced in 70-80% as compared to the original A*.

Dynamic Highway-Node Routing. Highway-Node Routing is a method that was firstly developed for static shortest path but then generalized for a dynamic setting as described in Schultes & Sanders [80]. Selection of *highway-nodes* is performed using the 'importance' of the nodes in which nodes used by many shortest paths will generate sparse graphs of the high-hierarchy network. At publishing time (2007), the authors claimed to have "the most efficient method in space requirement for the preprocessed data, while allowing query times several thousand times faster than Dijkstra's algorithm". They have developed variants for two kinds of processing scenarios: server and mobile. In the former scenario, an update is performed from the nodes affected in the higher level graph, when costs change in the edges. In the latter scenario, the set of potentially unreliable nodes is determined once changes in the cost occur; then if the search reaches a node to which a reconstruction should be performed (i.e., it is affected by changes), it continues searching in a sufficiently low level to guarantee that correct paths are still found. They conclude by stating that their method can handle effectively situations of dynamic changes in cost such as traffic jams, but also that a time-dependent variation is an important open issue.

Landmark-based (ALT) Dynamic Routing. It was present in Delling & Wagner [27] and two scenarios of dynamics are dealt with: Dynamic ALT and

Time-dependent ALT. The former allows changes in the costs of the edges, but these costs are time-independent—i.e, they are constant while there is no change. The latter assumes that costs are known functions that model changes depending on the time of the day the traversal is made in the edge. It turns out that costs are unlikely to ever drop from the cost of an “empty road” (i.e., traverse is made in full flowspeed) for all the normal cases of changes in costs (e.g., traffic jams and construction sites); thus ALT algorithm will always maintain an underestimation of cost as long as the initial estimations is made considering “empty roads”, and the algorithm is still able to provide optimal routes when costs rise though at the eventual cost of a larger search space (i.e., potentially slower). The authors, then, provide two variants of the Dynamic ALT: *Eager*—preprocessed data of ALT is carefully (i.e., only in the affected area) updated whenever changes occur; and *Lazy*—preprocessing is only triggered when unlikely drops of costs are encountered. The second scenario is adapted by taking out the ALT's bidirectionality of search, since this is prohibitive for time-dependent routing, and only updating the estimations when (unlikely) drops of costs occur.

Time-dependent SHARC-Routing. Time-dependent is recognized as a very common scenario in roads network, to which traffic conditions are reasonably predictable [27, 89, 23]. SHARC was proposed by Bauer & Delling in [11] and enhanced in [12]. It is based on arc-flags that notify whether an edge lies in a shortest path to a node in a certain partition C_i in which the graph has been splitted. As the graph is also divided in multi-levels of hierarchy, therefore SHARC combines techniques from arc-flags with hierarchical approaches. Delling [23] presents an adaptation of the SHARC-algorithm to handle time-dependent networks. Two settings of the Time-dependent SHARC-routing are presented, with a clear trade-off between preprocessing and query time. The faster preprocessing setting (1 hour and a half for continental-sized network) returns queries in an average of 17.5 milliseconds, which can be reduced to bellow 5 milliseconds at cost of considerably more preprocessing—above 12 hours.

3.4 Dynamic Shortest Path in a database context

On database technology resides the development of Intelligent Transportation Systems, at least as means of storing relevant data for such a system. Our research is interested on solving TDSP problems in a database context, and several reasons can be given to have processing of data coupled to storing (refer to [4, 2]). That is a setting which seems to not have received much focus from research, as most of time-dependent (or dynamic) shortest path solutions not even mention a database as part of their structure. From the best of our knowledge, we refer to the work of Ding et al. [28] that do consider database, and suggest a storage data model for a time-dependent shortest path solution, but that only have it as means of storing and fetching of data for an external memory-based shortest path algorithm to perform. Yin et al. [87] explicitly refer to a database based shortest path algorithm, which has not quite the kind of dynamics we consider in our research, as they only deal with moving objects and a classical shortest path approach (i.e., distance-based).

Chapter 4

Tools for Dynamic Shortest Paths in a database context

In this chapter we describe the development phase of the research, starting to describe the kind of (TDSP) problems were tackled in this research. We then describe the chosen approaches to implement solutions to the TDSP requests as formulated. In summary, all needed implementation is described along with algorithms and details necessary to understand the approach used. This is all done in an implementation-independent form, which is further materialized in the implementation of the approach described here to the study case (Chapter 5) of this research.

4.1 A set of TDSP problems and definition

Though TDSP problems may be conceptualized as two fundamental problems (see Table 3.1), in practice, it may assume different *flavors* and complexity. From a simple one-to-one TDSP for a given departure or arrival time up to a much more complex *traveling salesman problem*[1] in a time-dependent network. We present dimensions that may be involved in formulating a TDSP problem in Table 4.1 and depict an example of increasing complexity TDSP problems formulations (at conceptual level) in Figure 4.1.

As it is infeasible to deal with all complexity levels of TDSP problems presented in Figure 4.1, in the time available for this research, we focus on two problem definitions: one that forms the base of TDSP requests, and a more complex one that has many practical applications.

Before we formalize the two specific problems, let us define the general TDSP problem. Formally, this is the problem of finding a *least travel-time* path through a time-dependent graph $G_T(V, E, W)$, with $V = v_i$ being a set of nodes, $E \subseteq V \times V$ a set of edges and W a set of positive-valued functions. Associated

[1]Traveling salesman is a well-defined shortest path problem formulation where the goal is to visit a number of points in the graph in an optimal way, meaning that in practice the order in which the points to be visited along with the path through them are to optimized. By definition, the costs in the graph are travel times. We refer to Bondy & Murty (1976) [16] for a formal definition of the problem.

Table 4.1: Dimensions on TDSP problem requests.

Dimension	Attributes	Remark
Routing vehicle	Individual or Multiple (fleet)	
Planning horizon	A Priori or Immediate	immediate - en route optimization
Depot strategy	Closed or Open	open - not coming back to depot
Visit planning	Single or Multiple (set of visits)	
Order of visit	Fixed or Flexible	flexible - optimization of sequence also necessary
Departure/Arrival time	Fixed or Flexible	flexible - optimization of departure time also necessary
Time horizon (departue/arrival)	Constrained or Unconstrained	unconstrained - just in this case, optimization of departure/arrival time is completely flexible

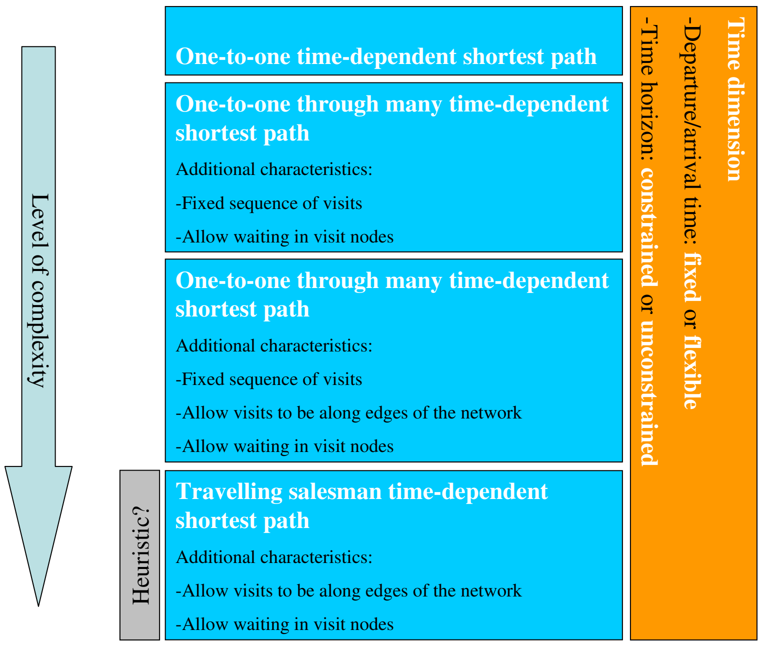

Level of complexity
Heuristic?
One-to-one time-dependent shortest path
One-to-one through many time-dependent shortest path
Additional characteristics:
-Fixed sequence of visits
-Allow waiting in visit nodes
One-to-one through many time-dependent shortest path
Additional characteristics:
-Fixed sequence of visits
-Allow visits to be along edges of the network
-Allow waiting in visit nodes
Travelling salesman time-dependent shortest path
Additional characteristics:
-Allow visits to be along edges of the network
-Allow waiting in visit nodes
Time dimension
-Departure/arrival time: fixed or flexible
-Time horizon: constrained or unconstrained

Figure 4.1: TDSP problem formulations in increasing complexity levels.

with every edge $(v_i, v_j) \in E$, there is a function $w_{i,j}(t) \in W$ for a time variable t in a time domain called an *edge-delay function*. This means that for any time point in T, one may know the cost to traverse a certain edge in the graph at that moment. Costs are usually given in delays, i.e. travel time to go from v_i to v_j in the graph.

4.1.1 A simple one-to-one TDSP for a given departure time

The most simple problem formulation for a TDSP request that one may imagine is to search for the optimal path between an *origin* (or source) and *destination* point for a certain time of departure. This problem definition, though quite trivial, has practical applications, such as to guide a vehicle en route with the "intelligence" of the time-dependent network being considered.

This problem formulation takes the general definition of TDSP problem (see Section 4.1) and searches the path starting from origin at a certain point in T. For the sake of clarity, from this point onwards in the thesis, we refer to this problem definition as TDSP-GDT, as a short for ***T****ime-****D****ependent* ***S****hortest* ***P****ath for a* ***G****iven* ***D****eparture* ***T****ime*.

4.1.2 One-to-one TDSP for a given interval of departure time

In practical applications, the time of departure of a vehicle may have some flexibility and thus we may want not only to search for a shortest path for a given point in time, but rather a path in which the travel time is minimal within a given time interval. In this context, to the complexity of finding a shortest path problem is added the need for optimizing the time of departure in such a way that the travel time is the minimal possible in the given interval.

Now, we want to find the *least total travel time* within a given interval $T = [ts, te]$ and also want to find out what is the path that uses up least travel time. In the literature, this type of request is called LTT (least travel time) [28], thus we refer to this problem formulation as TDSP-LTT, from ***T****ime-****D****ependent* ***S****hortest* ***P****ath with* ***L****east* ***T****ravel****T****ime in a given interval*.

4.2 Relevant concepts and definitions for solving TDSP problems

Before presenting a *defacto* approach to solve TDSP problems as formulated above, there are a few concepts and definitions that must be formalized. We start by formalizing the concept of *dynamics* in shortest path computation, by considering a simple situation of a directed graph with only two nodes and two possible ways (edges) connecting one point with another. This situation is depicted in Figure 4.2, along with the intuition of finding the time-dependent shortest path function as the minimum function comparing the two possible ways. In the formalization, both t and a function of t denote absolute time. A formal definition of the concept is:

- Let there be two functions $f(t)$ and $g(t)$ that define the time-dependency of two paths, such as arrival time in the target node of the path;

- Clearly, both functions are defined by the departure time from the origin node of the path t accumulated by the cost associated to traverse the given edge, or more formally $f(t) = t + w_{f(t)}(t)$;
- Consequently the definition of the time-dependent shortest path is determined by a minimal function comparing the functions $f(t)$ and $g(t)$ of the two possible ways in this case, therefore a function $h(t)$ that has the following definition:

$$h(t) = \begin{cases} f(t) & \text{if } f(t) \leq g(t) \\ g(t) & \text{otherwise} \end{cases}$$

- This means that depending on the chosen departure time t, may $f(t)$ be the desired path function (minimal), otherwise $g(t)$, and characterizes the "dynamics" of the shortest path in the context of TDSP problems.

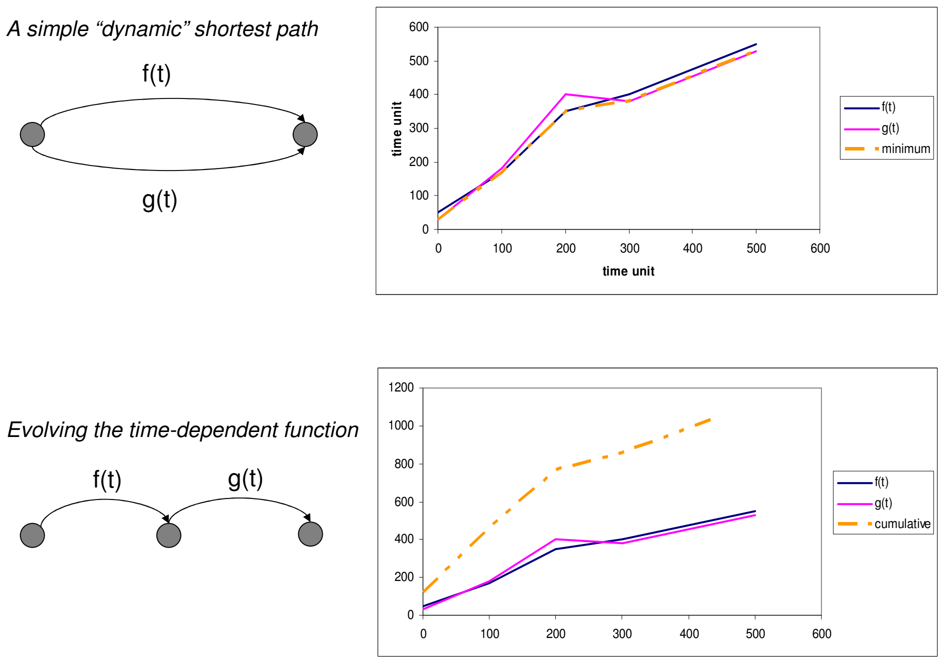


Figure 4.2: Intuition and example of relevant concepts for solving TDSP problems.

Now, suppose we add one more node to the graph, but only consider one possible path between origin and destination, passing through an intermediate node. The issue then is how to progress the time refinement of the time-dependent functions, and the intuition of this concept is shown in Figure 4.2. A formalization of this concept is:

- Let there be two functions $f(t)$ and $g(t)$ which define the time-dependency of the two parts that define the path origin to destination;
- Notice that $f(t)$ and $g(t)$ can actually even be two functions $\in W$, describing the delays (i.e., relative time) for traversing each part of the path, but can also be an absolute time function;
- If we now want to know a cumulative function of $f(t)$ and $g(t)$, thus define a single function $h(t)$ that will describe the time-dependency of the path from origin to destination, we define $h(t)$ as the cumulative summation of $f(t)$ and $g(t)$. We call cumulative summation because it is not a simple matter of summing-up $f(t)$ and $g(t)$, since one needs to be aware that for the intermediate node to be reached, and $g(t)$ to be considered, a certain amount of time given in $f(t)$ was used up;
- Therefore, the formal definition of the cumulative function is $h(t) = f(t) + g(t + f(t))$, or intuitively: if one departs from origin at time t, it will take $f(t)$ to arrive in the intermediate node, thus it will only depart from this node at time $t + f(t)$, taking $g(t + f(t))$ to get to the target node. For the case the functions denote absolute time, the definition is generalized to $h(t) = f(t) + g(f(t))$.

These two formal definitions above constitute the core of the necessary time refinement for solving TDSP problems. They identify how the time-dependent function evolves as the problem evolves, as well as how to determine the minimal function that "hides" in itself the dynamics of the shortest path (i.e., if one leaves at t, what way to use). The *defacto* solution for TDSP that is introduced below, has these two concepts.

4.3 Solving a set of TDSP problems

In its static counterpart, a shortest path problem commonly applies the good old algorithm by Dijkstra in 1959 [16], or some variation of it that makes use of heuristics to reduce the search space [85]. As originally presented, Dijkstra's algorithm has the characteristic of always providing an optimal solution by searching the whole graph.[2] On the other hand, applying heuristics to reduce the search space may compromise optimality for the sake of providing faster computation of shortest paths.

For our purpose of solving TDSP in a database context, we decided to make use of the Dijkstra-based algorithm presented in Ding et al. [28], in which the authors prove that their approach outperforms others in computational tests for TDSP-LTT types of query (i.e., our most complex problem definition). But this can also be generalized for the simple TDSP-GDT, by simplifying the arrival-time function to be a single-point function (or a simple pair of values $(t, g(t))$—i.e., departure time *vs* arrival time—with departure time being the given time

[2]Its original goal is to find one-to-all shortest paths, thus it needs to search the whole graph, but this does not necessarily hold for the usual variation of Dijkstra's for solving one-to-one shortest path.

of the request. In summary, the novelty of the solution is in the decoupling of time refinement and path selection, meaning that the path only is determined after the time dimension has been refined to the nodes in the graph, with so-called *arrival-time functions* that display departure time from source against arrival time in the node. The path-selection follows by matching arrival time in predecessor node plus edge-delay (cost) to be traversed with arrival time of outgoing node, in a backwards manner. From this decoupling comes the name of the algorithm: Two-step LTT.

4.3.1 Explaining the Two-step LTT approach

In general, the Two-step LTT algorithm was presented and described in Section 3.2.4, but here we detail it in such a way that one should be able to implement it for TDSP problems. For this reason, we adopt an approach of gradually explaining the algorithm and its conceptualization. We follow this description by identifying what are the tools that need to be implemented so that the approach can work correctly.

First step—Dijkstra based time-refinement. It intends to define earliest arrival-time functions $g_i(t)$ for all nodes v_i in the graph, expanding from the source v_s just like Dijkstra's algorithm (Figure 4.4). For the terminology used in the approach, we refer to Section 3.2.4. Arrival-time functions are incrementally refined in the given departure time interval $T = [ts, te]$. The algorithm starts with an initialization process within lines 01-03, in which $g_s(t) \leftarrow t \quad \text{for}$ $t \in T; \tau_s \leftarrow t_s$ and $g_i(t) \leftarrow \infty \quad \text{for} \quad t \in T; \tau_i \leftarrow t_s$, i.e. the earliest arrival-time function for the start node is (arrival time equals departure time), and is undefined (∞) for other nodes in the graph. A priority queue Q is then formed by pairs of values of departure time and arrival time $(\tau_i, g_i(t))$, ordered by $g_i(t)$, for all the nodes in line 04. The time-refinement starts by dequeueing the top pair $(\tau_i, g_i(t))$ of Q, which by principle will be the pair $(ts, g_s = ts)$ from the start node, but also to acknowledge that the next earliest arrival-time function pair in Q at top of the queue $(\tau_k, g_k(t))$. The expansion in the graph continues in lines 09-10 by identifying a subinterval $I' = [\tau_i, \tau_i']$ that depends on the least edge-delay value coming to the dequeued node v_i at the time of the top function value $g_k(t)$, as it is in $\Delta \leftarrow min\{w_{f,i}(g_k(\tau_k)) \,|(v_f, v_i) \in E\}$. The computation of Δ allows to identify the next earliest possible arrival time via any edge (v_i, v_j) as $g_k(\tau_k) + \Delta$ due to the FIFO property of G_T, and it is followed by the computation of the I' that can be proven to be the interval in which the arrival-time function $g_i(t)$ is already well-defined (i.e., it is already the minimal possible) [28]. Values in the priority queue Q with τ_i within $I' = [\tau_i, \tau_i']$ for v_i node have then no reason to be in Q anymore, though that is not explicitly presented in the published algorithm. Now, with the acknowledgment that $g_i(t)$ is well-defined in $I' = [\tau_i, \tau_i']$, we can use $g_i(t) \quad \text{for} \quad t \in [\tau_i, \tau_i']$ to update arrival-time functions $g_j(t)$ of outgoing neighbours v_j of v_i, which happens in lines 11-14. That is done in $g_j'(t) \leftarrow g_i(t) + w_{i,j}(g_i(\tau_i))$, again, only where t is within $I' = [\tau_i, \tau_i']$. The concept here is simple: one is at time $g_i(t)$ in node v_i and wants to go to v_j, traversing

$(v_i, v_j) \in E$ with cost $w_{i,j}(g_i(\tau_i))$[3] (i.e.. the cost for leaving at the moment one is at v_i). Updated $g'_j(t)$ where t is within $I' = [\tau_i, \tau'_i]$ is compared to what is already in $g_j(t)$, getting the minimal function $g_j(t) \leftarrow min\{g_j(t), g'_j(t)\}$ for $\in [\tau_i, \tau'_i]$ and leaving the rest of values in $T = [ts, te]$ as they were. New values of $g_j(t)$ where t is within $I' = [\tau_i, \tau'_i]$ can then be updated in Q ($update\,(Q, (\tau_j, g_j(t)))$). Condition of termination of the algorithm (lines 16-20) is either by finishing the priority queue Q (i.e., **while** $|Q| \geq 2$), or by having the full interval $T = [ts, te]$ well-defined (i.e., $\tau_i \leftarrow \tau'_i$; **if** $\tau_i \geq t_e$) for the target node v_e (i.e., **if** $v_i = v_e$). In this latter case, we are already assured that $g_e(t)$ for the target node v_e is well-defined for the whole given interval of request $T = [ts, te]$. Therefore there is no reason for continuing iterating over the graph's nodes. If that is not the yet case, the pair $(\tau_i = \tau'_i, g_i(t))$ is enqueued in Q ($enqueue\,(Q, (\tau_i, g_i(t)))$) as it needs further refinement (i.e., arrival-time function where $t \geq \tau_i = \tau'_i$ is not found to be minimal yet). A small running example of the core of the time-refinement algorithm, namely the part in which the outgoing nodes' arrival-time functions $g_j(t)$ are updated is illustrated in Figure 4.5. After finishing the time-refinement step, the best starting time can be determined using the arrival-time function of the target node v_e and finding the least travel time as the minimal argument of arrival time (y-axes) in v_e minus the departure time from the source (x-axes) ($t^* \leftarrow argmin_{t \in T}\{g_e(t) - t\}$) in the main algorithm call in line 03 (Figure 4.3).

```
Algorithm Two-Step-LTT($G_T(V, E, W), vs, ve, T$)
01: $\{g_i(t)\} \leftarrow timeRefinement(G_T, v_s, v_e, T)$;
02: if $\neg$(g$_e(t) = \infty$ for the entire $[t_s, t_e]$ then
03:     $t^* \leftarrow argmin_{t \in T}\{g_e(t) - t\}$;
04:     $p^* \leftarrow pathSelection(G_T, \{g_i(t)\} vs, ve, t^*)$;
05:     return $(t^*, p^*)$;
06: else return $\emptyset$;
```

Figure 4.3: Outline of Two-step least travel-time algorithm (28) .

Second step—Fast path-selection. An optimal $v_s - v_e$ path is computed in a backward manner determining the predecessor node, starting from the destination node v_e (Figure 4.6). Based on the arrival-time functions $g_i(t)$ and the optimal starting time t^* the predecessor node of any v_j can be determined matching the condition $g_i(t^*) + w_{i,j}(g_i(t^*)) = g_j(t^*)$ (line 05). That holds under the assumption that there is no waiting time in the nodes [28]. Thus, one can find the predecessor node by adding to the arrival-time function $g_i(t^*)$ of v_i the edge-delay from v_i to v_j (i.e., $w_{i,j}(g_i(t^*))$) and finding which edge leads to the arrival-time function value for the optimal starting time in v_j (i.e., $g_j(t^*)$). Figure 4.7 shows an example of path-selection.

[3]It should be trivial to realize that $g_i(t)$ has in it the concept of $g_i(t + f(t))$ where $f(t)$ is the arrival-time function $g_s(t)$ in the source node v_s. Thus $w_{i,j}(g_i(t))$ is a generalization of the concept of evolving the time-dependent function as explained in Section 4.2

```
**Algorithm** *timeRefinement*$(G_T(V,E,W), v_s, v_e, T)$
01: $g_s(t) \leftarrow t$ for $t \in T; \tau_s \leftarrow t_s$;
02: for each $v_i \neq v_s$ do
03:     $g_i(t) \leftarrow \infty$ for $t \in T; \tau_i \leftarrow t_s$;
04: Let $Q$ be a priority queue initially containing pairs, $(\tau_i, g_i(t))$,
05: for all nodes v$_i$v$_i \in V$, ordered by $g_i(\tau_i)$ in ascending order;
06: while $|Q| \geq 2$ do
07:     $(\tau_i, g_i(t)) \leftarrow dequeue(Q)$;
08:     $(\tau_k, g_k(t)) \leftarrow head(Q)$;
09:     $\Delta \leftarrow min\ \{w_{f,i}(g_k(\tau_k))\,|(v_f, v_i) \in E\}$ ;
10:     $\tau_i^{'} \leftarrow max\{t\,|\ g_i(t) \geq g_k(\tau_k) + \Delta\}$;
11:     for each $(v_i, v_j) \in E$ do
12:         $g_j^{'}(t) \leftarrow g_i(t) + w_{i,j}(g_i(\tau_i)$ for $t \in \left[\tau_i, \tau_i^{'}\right]$;
13:         $g_j(t) \leftarrow min\{g_j(t), g_j^{'}(t)\}$ for $\in \left[\tau_i, \tau_i^{'}\right]$;
14:         $update\,(Q, (\tau_j, g_j(t)))$;
15:     $\tau_i \leftarrow \tau_i^{'}$;
16:     if $\tau_i \geq t_e$ then
17:         if
18:             return $\{g_i(t)\,|v_i \in V\}$;
19:     else
20:         $enqueue\,(Q, (\tau_i, g_i(t)))$;
21: return $\{g_i(t)\,|v_i \in V\}$;
```

Figure 4.4: Ding et al. (2008) (28) time refinement algorithm.

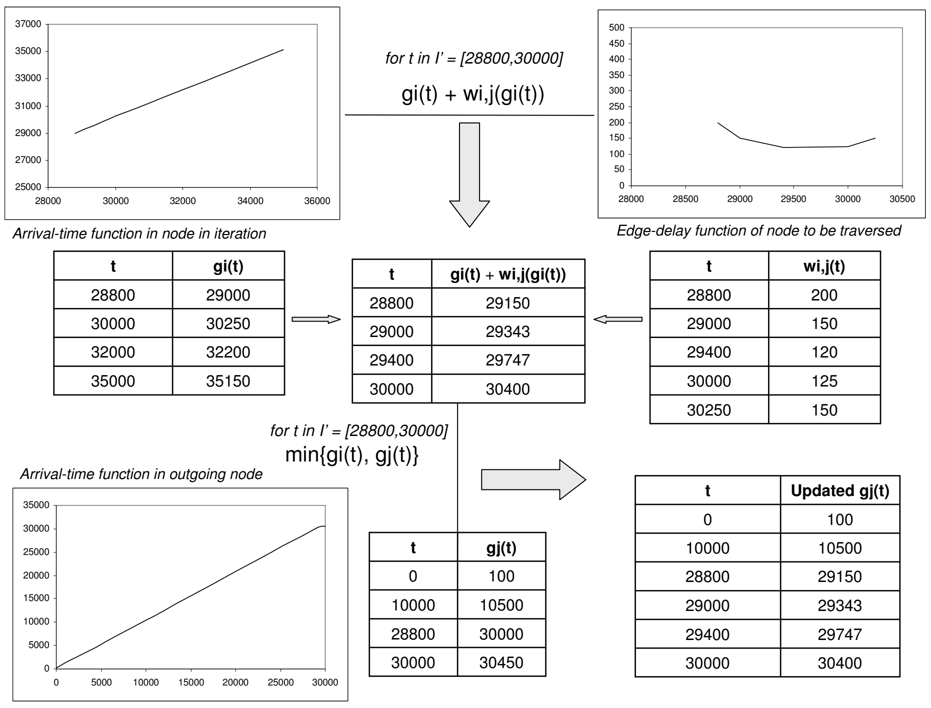
4.3.2 Identifying tools to be implemented for the Two-step LTT approach

Any algorithm to be implemented may require the development of auxiliary functions to correctly do what it is supposed to do. In the case of Two-step LTT approach these required functions are primarily associated with the first step of time-refinement, in which the crucial part to identify a time-dependent shortest path occurs. Looking at the time-refinement algorithm (Figure 4.4) one may identify four pieces of code that constitute the approach:

1. **Initialization:** creation of initial arrival-time functions up to the preparation of the priority queue.

2. **Determination of subinterval to update:** which starts by getting the two top of queue pairs, followed by the determination of parameter Δ, then determination of the subinterval $I' = [\tau_i, \tau_i']$ where the arrival-time function of the node in iteration is well-defined (i.e., minimum possible) and thus it can be used to update the arrival-time functions of outgoing nodes.

3. **Update of outgoing nodes arrival-time functions:** this is the core of the algorithm; on the basis of arrival-time function of the node in itera-


for t in I' = [28800,30000]
gi(t) + wi,j(gi(t))
Arrival-time function in node in iteration
Edge-delay function of node to be traversed
for t in I' = [28800,30000]
min{gi(t), gj(t)}
Arrival-time function in outgoing node

t	gi(t)
28800	29000
30000	30250
32000	32200
35000	35150

t	gi(t) + wi,j(gi(t))
28800	29150
29000	29343
29400	29747
30000	30400

t	wi,j(t)
28800	200
29000	150
29400	120
30000	125
30250	150

t	gj(t)
0	100
10000	10500
28800	30000
30000	30450

t	Updated gj(t)
0	100
10000	10500
28800	29150
29000	29343
29400	29747
30000	30400

Figure 4.5: Running example of updated of arrival-time function.

tion and the edge-delay function of the eventual edge to be traversed, one can update the arrival-time function of outgoing neighbour nodes in the subinterval determined in the previous part of the code.

4. **Checking of termination condition:** this is meant to check whether the termination condition is met, but will also enqueue a pair of values into the queue, in case the termination condition is not met yet.

Table 4.2 summarizes what are the tools needed for each of the four parts of the time-refinement algorithm of Two-step LTT, which will eventually trigger an auxiliary function implementation apart from the main code. This is followed by the description of these auxiliary function definitions in the next section.

4.4 Auxiliary tools for the Two-step LTT approach

Here, we present the definition of auxiliary tools for the Two-step LTT approach that we found relevant or necessary. As we want it to be implementation-independent, what is shown is what the algorithm should do, at conceptual

```
Algorithm pathSelection($G_T(V, E, W)$,$\{g_i(t)\}v_s, v_e, t^*$)
01: $v_j \leftarrow v_e$;
02: $p^* \leftarrow \emptyset$;
03: while  $v_j \neq v_s$  do
04:     for each  $(v_i, v_j) \in E$  do
05:         if  $(g_i(t^*) + w_{i,j}(g_i(t^*)) = g_j(t^*)$  then
06:             $v_j \leftarrow$
07:     $p^* \leftarrow (v_i, v_j) \cdot p^*$;
08: return p*;
```

Figure 4.6: Ding et al. (2008) (28) path selection algorithm.

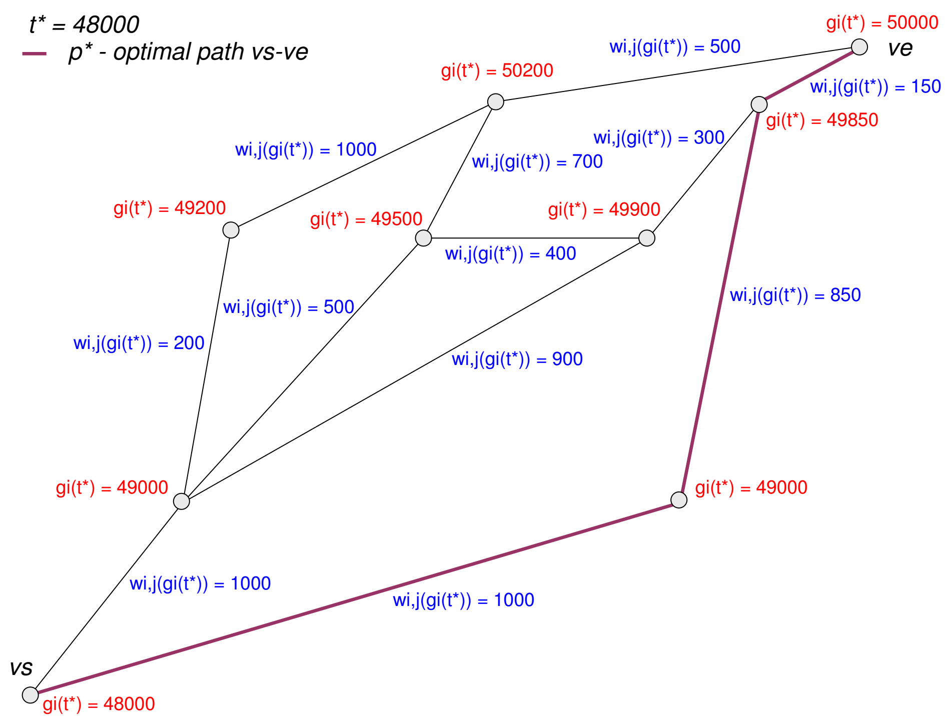


Figure 4.7: Running example of an optimal path-selection p^* for the optimal starting time t^*.

level, and where possible, pseudo-code that describes mathematically the tool to be implemented.

4.4.1 Initialization tool: generate initial arrival-time functions

The first step to be carried out in the time refinement algorithm of Two-step LTT approach is to initialize the arrival-time functions $g_i(t)$ for all the nodes in the graph. This was discussed in the previous section and the code for the tool

Table 4.2: Summary of tools for the development of the Two-step LTT approach.

Algorithm part	**Required tools**
Initialization	May require a function to initialize the arrival-time functions for the source node and all the other nodes in the graph; Mechanism for creating the priority queue by enqueing all pair values from the arrival-time functions previously initialized.
Determination of subinterval to update	Some sort of x-value look-up (or interpolation, depending on implementation choices) in a mathematical function for finding the least edge-delay value coming into a node; Some sort of y-value look-up (or interpolation, depending on implementation choices) in a mathematical function for determining the upper bound τ_i' for the subinterval to update I'.
Update of outgoing nodes' arrival-time functions	Function to add arrival-time function of node in iteration to the edge to be traversed to reach an outgoing node (update arrival-time function of outgoing node); Function to compare updated arrival-time function of outgoing node with what is already in arrival-time function, getting the minimal arrival-time function in the comparison (look-up functions mentioned in the previous part of the code may come to be useful in this context); Mechanism of updating the priority queue with updated arrival-time function values for the outgoing nodes.
Checking of termination condition	Only a single value pair update to the queue may be needed in this part of the code.

is derived from lines 01-03 of the algorithm in Figure 4.4. Our adaptation is presented in Figure 4.8.

```
Algorithm generateInitialization($G_T(V,E,W), v_s, T=[t_s,t_e]$)
for each $v_i \in V$ do
    if $v_i = v_s$ then
        $g_s(t) \leftarrow t$ for $t \in T$;
    else
        $g_i(t) \leftarrow \infty$ for $t \in T$;
return $\{g_i(t)\,|v_i \in V\}$;
```

Figure 4.8: Initialization algorithm.

4.4.2 Tools for X-/Y-value look-up

In the time refinement algorithm (Figure 4.4), there are situations in which the code needs to know a certain x- or y-value of a mathematical function, either from an arrival-time function or an edge-delay function, not necessarily from one of the pairs representing the given function. A look-up function to determine an axis value given a value of another axis is demanded. How to present such a function in a pseudo-code way, depends on implementation choices, particularly on how one represents those mathematical functions. For instance, if these functions are represented as pairs of values in an array, or even as table in a database, to determine an eventual intermediate point (i.e., a x-/y-value for which the given y-/x-value is not one of the pairs that describe the function) an interpolation is needed. Now, for a more didactical representation, let's imagine these functions are represented in a geometric plan (i.e., a XY cartesian plane). In this case, a geometric interpolation draws a straight line with given x- or y-value in one variable, with the other variable tending to ∞. Then, we take the intersection point for the two geometries (the mathematical function and the straight line) that contains the desired x-/y-value. Obviously, this only works in the case of monotonic function.[4] By definition, arrival-time functions are always of this nature, while edge-delay functions may or may not be, depending on how they are conceptualized. If we assume this monotonic function nature, one can create such look-up functions as in pseudo-code in Figure 4.9. We adopt the convention of using t for the x-axis as these are always "time" in our context.

```
Algorithm $lookUp\_Yvalue(f(t), x)$
$line \leftarrow makeLine(makePoint(x, -\infty), makePoint(x, \infty))$;
$point \leftarrow intersection(f(t), line)$;
return $\{getX(point)\}$;

Algorithm $lookUp\_Xvalue(f(t), y)$
$line \leftarrow makeLine(makePoint(-\infty, y), makePoint(\infty, y))$;
$point \leftarrow intersection(f(t), line)$;
return $\{getY(point)\}$;
```

Figure 4.9: Look-up algorithms.

4.4.3 Tool to add-up two mathematical functions

In the context of the time refinement algorithm (Figure 4.4) of the Two-step LTT approach, the addition of two mathematical functions occurs when the code seeks to update the arrival-time function of an outgoing node. This is done on the basis of the arrival-time function of the node in iteration added to the edge-delay function of the link to be traversed in the graph. Since the edge-delay

[4]One which preserves the given order, either only increasing or decreasing (i.e., none step or backs and forwards in the function.

functions can be not constant, this addition need to be performed on a point-by-point basis. As the time refinement in the approach normally happens in subintervals $I' = [\tau_i, \tau_i']$ of the given interval of departure time request $T = [ts, te]$, the addition will usually affect only part of the mathematical function. A pseudo-code for implementing such a tool can be seen in Figure 4.10.

```
Algorithm addUpTwoFunctions($g_i(t), w_{i,j}(t), I' = [\tau_i, \tau_i']$)
$pointarray \leftarrow union(serializePoints(g_i(t)), serializePoints(w_{i,j}(t)))$;
for $p \in pointarray$ do
    if $getX(p) <= \tau_i$ and $getX(p) <= \tau_i'$ then
        $p' \leftarrow lookUp\_Yvalue(g_i(t), getX(p)) +$
        $lookUp\_Yvalue(w_{i,j}(t), lookUp\_Yvalue(g_i(t), getX(p)))$;
        $update(pointarray, p')$;
$g_j'(t) \leftarrow makeLineFromArray(pointarray)$;
return $\{g_j'(t)\}$;
```

Figure 4.10: Add-up two functions algorithm.

4.4.4 Tool to determine the minimal function from two mathematical functions

After performing the addition of two mathematical functions (Section 4.4.3), namely the arrival-time function of the node in iteration added to the edge-delay function of the link to be traversed in the graph, this updated added mathematical function has to be compared with the existing arrival-time function in the outgoing node. Again, as this comparison will usually apply for a certain subinterval $I' = [\tau_i, \tau_i']$ of the given interval of departure time request $T = [ts, te]$, it is not only a matter of getting the minimal function between the two being compared. It can be performed on a point-by-point basis (Figure 4.11).

```
Algorithm minimalOfTwoFunctions($g_j(t), g_j'(t), I' = [\tau_i, \tau_i']$)
$pointarray \leftarrow union(serializePoints(g_j(t), serializePoints(g_j'(t))$;
for $p \in pointarray$ do
    if $getX(p) \geq \tau_i$ and $getX(p) \leq \tau_i'$ then
        $p' \leftarrow makePoint(getX(p), ming_j(getX(p)), g_j'(getX(p)))$;
        $update(pointarray, p')$;
$g_j(t) \leftarrow makeLineFromArray(pointarray)$;
return $\{g_j(t)\}$;
```

Figure 4.11: Determination of the minimal function algorithm.

4.5 Two-step LTT approach for solving a set of TDSP problems

At this point, following the previous sections (4.3,4.4) it should be clear how one can solve TDSP problems by applying the Two-step LTT approach, particularly for the case of a TDSP-LTT problem, for which the approach was originally proposed. For the sake of clarity, especially considering that we have defined several auxiliary functions, we now present again the Two-step LTT approach but modified with the inclusion of the auxiliary functions.

By generalizing the Two-step LTT approach to solve TDSP-GDT (i.e., with a fixed departure time) part of the code for time refinement (Figure 4.12) becomes useless, as there is no interval of departure time involved. Also, the path selection step can also be tuned-up sorting arrival time values ascendently before checking the condition of identification of the predecessor node in the path. This must happen when all the possible predecessor nodes in the path are well-defined (i.e., with least arrival-time function), after reaching the target node, thus the condition $(g_i(t^*) + w_{i,j}(g_i(t^*)) = g_j(t^*)$ will hold for any possible predecessor node. Figure 4.13 presents the Two-step LTT approach generalized to solve TDSP-GDT problems.

4.6 Interlude: encompassing multimodal shortest path problems—a conceptual level discussion

4.6.1 Why only at conceptual level?

In a realistic logistics and transportation setting, reasons may exist for considering an multimodal chain (see [74]). By a multimodal transport chain, we mean transport involving multiple transport modes: trucks and trains, for instance. The complexity added by such transport is caused by additional timing constraints (e.g., time-windows) and delays when switching modes. Meaning that exchange of modes is not simply a matter of existence of topological connections, but that this meeting is also constrained in the time dimension [82]. More precisely, the time dimension is considered in absolute sense, meaning that it has to be possible to change to another mode at a specific moment. What's more, this may also imply delays or extra costs for changing mode, which may influence the shortest path definition as well.

As stated in Chapter 1, a multimodal network setting may be part of a shortest path problem investigation as the one treated in this research, but some motives lead us to finally not consider this setting in our development, particularly at the implementation level of it. The first reason for this choice is the realization that single mode TDSP problems already have "enough" complexity, which can, of course, depend on how one formulates the problems. In our research, we have defined a rather complex TDSP problem, namely TDSP-LTT, as part of our set of problems to be tackled, hence one may think that we only wanted to be safe in not adding the complexity of a multimodal network for solving TDSP. Furthermore, it is fair to mention that there was no availability

```
Algorithm 1 Two-Step-LTT$(G_T(V,E,W), vs, ve, T)$
$\{g_i(t)\} \leftarrow timeRefinement(G_T, v_s, v_e, T)$;
if $\neg$(g$_e(t) = \infty$ for the entire $[t_s, t_e]$ then
    $t^* \leftarrow argmin_{t \in T}\{g_e(t) - t\}$;
    $p^* \leftarrow pathSelection(G_T, \{g_i(t)\} vs, ve, t^*)$;
    return $(t^*, p^*)$;
else return $\emptyset$;
```

```
Algorithm 2 timeRefinement$(G_T(V,E,W), v_s, v_e, T)$
$generateInitialization(G_T, v_s, T); \tau_i \leftarrow t_s$;
Let $Q$ be a priority queue initially containing pairs, $(\tau_i, g_i(t))$,
for all nodes v$_i$v$_i \in V$, ordered by $g_i(\tau_i)$ in ascending order;
while $|Q| \geq 2$ do
    $(\tau_i, g_i(t)) \leftarrow dequeue(Q)$;
    $(\tau_k, g_k(t)) \leftarrow head(Q)$;
    $\Delta \leftarrow min\ \{lookUp\_Yvalue(w_{f,i}(t), g_k(\tau_k))\,|(v_f, v_i) \in E\}$ ;
    $\tau_i' \leftarrow lookUp\_Xvalue(g_i(t), g_k(\tau_k) + \Delta)$;
    for each $(v_i, v_j) \in E$ do
        $g_j'(t) \leftarrow addUpTwoFunctions(g_i(t), w_{i,j}(t), I' = [\tau_i, \tau_i'])$;
        $g_j(t) \leftarrow minimalOfTwoFunctions(g_j(t), g_j'(t), I' = [\tau_i, \tau_i'])$;
        $update\ (Q, (\tau_j, g_j(t)))$;
    $\tau_i \leftarrow \tau_i'$;
    if $\tau_i \geq t_e$ then
        if
            return $\{g_i(t)\,|v_i \in V\}$;
    else
        $enqueue\ (Q, (\tau_i, g_i(t)))$;
return $\{g_i\,|v_i \in V\}$;
```

```
Algorithm 3 pathSelection$(G_T(V,E,W), \{g_i(t)\} v_s, v_e, t^*)$
$v_j \leftarrow v_e$;
$p^* \leftarrow \emptyset$;
while $v_j \neq v_s$ do
    for each $(v_i, v_j) \in E$ do
        if $lookUp\_Yvalue(g_i(t), t^*) +$
        $lookUp\_Yvalue(w_{i,j}(t), lookUp\_Yvalue(g_i(t), t^*)) =$
        $lookUp\_Yvalue(g_j(t), t^*)$ then
            $v_j \leftarrow v_i$; break;
    $p^* \leftarrow (v_i, v_j) \cdot p^*$;
return p*;
```

Figure 4.12: Two-step LTT approach (28) with adapted presentation to include auxiliary tools developed.

of a multimodal time-dependent network with real-world data. One may interpret it was infeasible to consider a multimodal TDSP as originally planned with real-world data. But more importantly, it is our believe that multimodal

```
Algorithm 1 Two-Step-LTT$(G_T(V, E, W), vs, ve, t)$
$\{g_i(t)\} \leftarrow timeRefinement(G_T, v_s, v_e, t)$;
if $\neg$(g$_e(t) = \infty$ for the entire $[t_s, t_e]$ then
    $p^* \leftarrow pathSelection(G_T, \{g_i(t)\} vs, ve)$;
    return $(p^*)$;
else return $\emptyset$;
```

```
Algorithm 2 timeRefinement$(G_T(V, E, W), v_s, v_e, t)$
$g_s) \leftarrow t; \tau_s \leftarrow t_s$;
for each $v_i \in V \neq v_s$ textbfdo    $g_i \leftarrow \infty$; $\tau_i \leftarrow t_s$;
Let $Q$ be a priority queue initially containing pairs, $(\tau_i, g_i)$,
for all nodes v$_i$v$_i \in V$, ordered by $g_i($ in ascending order;
while $|Q| \geq 2$ do
    $(\tau_i, g_i) \leftarrow dequeue(Q)$;
    $(\tau_k, g_k) \leftarrow head(Q)$;
    for each $(v_i, v_j) \in E$ do
        $g'_j \leftarrow g_i + lookUp\_Yvalue(w_{i,j}(t), g_i)$;
        $g_j \leftarrow min\{g_j, g'_j\}$;
        $update\,(Q, (\tau_j, g_j))$;
    if
        return $\{g_i\,|v_i \in V\}$;
    else
        $enqueue\,(Q, (\tau_i, g_i))$;
return $\{g_i \| \text{v}_i \in V\}$;
```

```
Algorithm 3 pathSelection$(G_T(V, E, W), \{g_i\} v_s, v_e, t^*)$
$v_j \leftarrow v_e$;
$p^* \leftarrow \emptyset$;
while $v_j \neq v_s$ do
    for each $(v_i, v_j) \in E$ ordered by $g_i$ do
        if $g_i + lookUp\_Yvalue(w_{i,j}(t), g_i) = g_j$ then
            $v_j \leftarrow$
    $p^* \leftarrow (v_i, v_j) \cdot p^*$;
return p*;
```

Figure 4.13: Two-step LTT approach (28) generalized to solve TDSP-GDT.

TDSP can at least be in part encompassed in a single mode TDSP by means of encoding smartly the edge-delay functions. This is discussed at conceptual level in the sequel of this chapter.

4.6.2 Multimodal TDSP in a single mode TDSP setting

In its perhaps most simple formulation, a multimodal TDSP has to consider that changing transportation mode implies incurred extra cost for this exchange. In a TDSP perspective, this extra cost means a certain time delay without actual moving, which applies either for a individual transportation or a cargo changing the transportation mode. Liu & Meng (2008) [56] treat multimodal shortest path by first identifying what they call *switch points*, which are nodes in the network in which a mode transition is possible. To extend this idea for a TDSP perspective, we suggest the accommodation of extra cost by means of repeating any multimodal node in as many single modal nodes exist. Different modal nodes would be connected by a *virtual* edge that only exists in the *time dimension*, with a delay function associated to describe the extra cost for the transition of mode. An illustration of this idea is provided in Figure 4.14, with two modes and a possible transition between them represented by an extra cost that is modeled by a "virtual" edge-delay function that connects the transition node repeated in two single modal nodes. A somewhat similar idea is presented in Bérubé et al. (2006) [15] for allowing travelers to wait at a certain node to take a later flight connection that eventually would be of benefit in the complete route along a set of points to be visited. In their case, clearly they purposely take away the FIFO assumption, as the waiting allowed is actually in there to eventually reach the subsequent node earlier by departing later. Our adaptation for encompassing multimodal routing in a single mode TDSP setting, on the other hand, may well still work under the FIFO assumption, by only realistically considering that an extra time may apply to a the change of mode, and not to take any travel time advantage by departure in a later time. In fact, TDSP solutions for FIFO graphs can accommodate general (i.e., non-FIFO) graphs by allowing waiting time in the nodes and transferring this waiting time to the edge-delay functions, or more formally by taking each edge-delay function $w'_{i,j}(t)$ in the non-FIFO graph and defining a $w_{i,j}(t) = \Delta_{i,j}(t) + w'_{i,j}(t)$, which would turn the graph into a FIFO graph [28].

It should be noticed that the same approach can also be adapted to model more complex time constraints in a multimodal TDSP, such as time-windows for changing the mode. This may be useful when the exchange of modes is only possible at certain intervals along the day. For instance, an impossibility of transition of mode between 8 and 16h can be represented by raising the delay value in the function by 4 hours of 'extra travel time' at 8h. To then let it to linearly decrease until it reaches back the usual transition of mode extra travel time (Figure 4.15).

Lately, the described approach would only require some data modeling and handling, not real changes in the solution of TDSP problems. At least a partial inclusion of the complexity of real-world multimodal routing could already be envisaged in such a way.

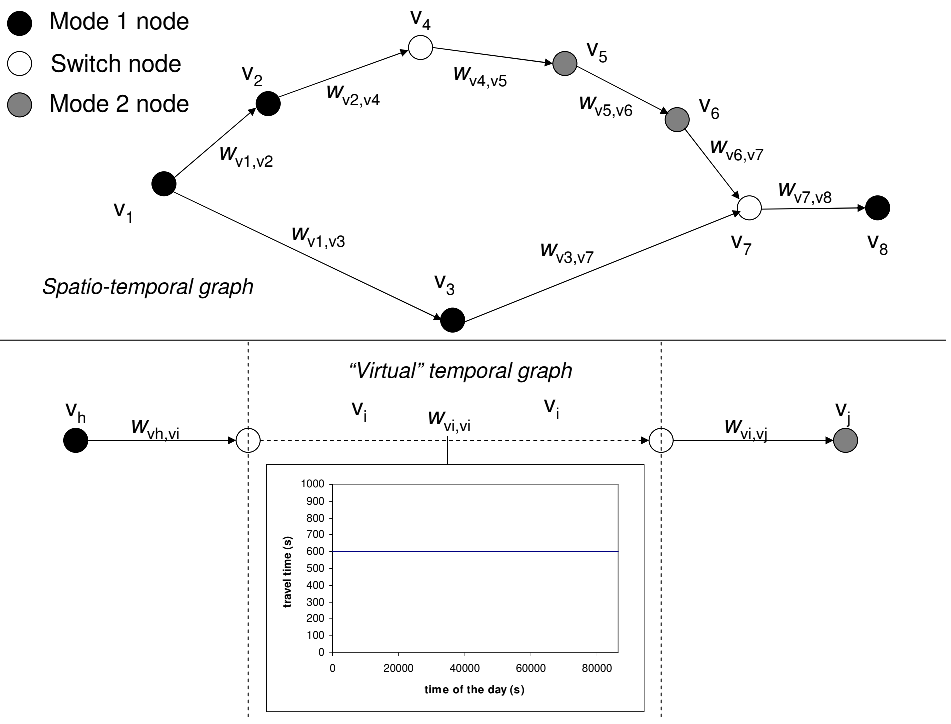


Figure 4.14: Allowing extra time for mode changes in a multimodal network by creating a "virtual" edge that connects the switch point to itself before reaching the subsequent node of a different mode.

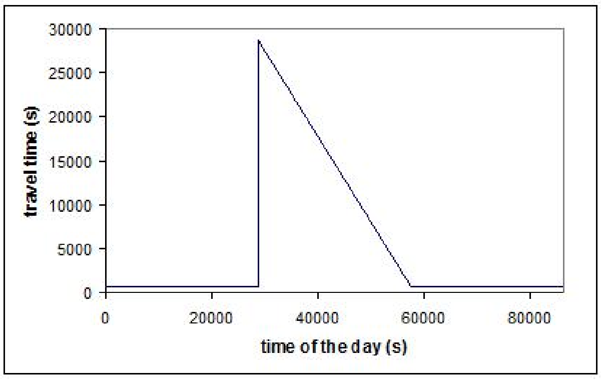


Figure 4.15: Representing time-windows in the edge-delay function of the transition of modes in a multimodal network.

Chapter 5

Case study implementation and performance tests

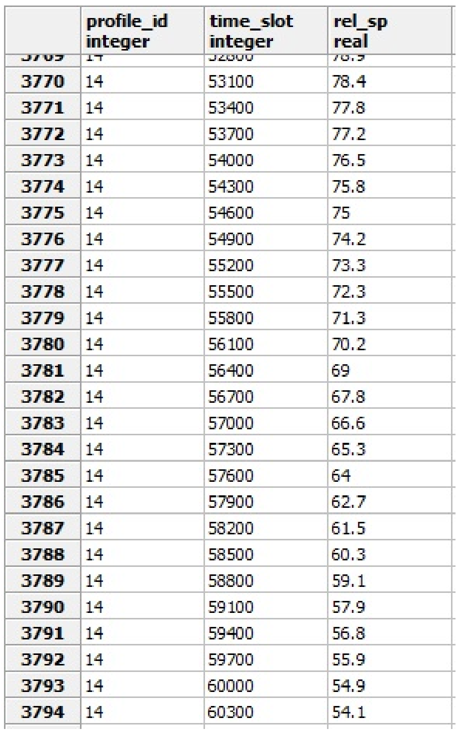
This chapter consists of "putting into practice" the solutions developed and presented in the previous Chapter 4. We start with a description of the real-world GIS dataset used for testing the implemented time-dependent shortest path solutions, and then present how we transform this GIS dataset into a graph, as the mathematical abstraction that the developed solutions need as input. We follow by identifying the computational limitations of the approach on preliminary running tests, as well as by presenting how one could optimize the computation and cope with (at least some of) these limitations. Finally, we set up a proof-of-concept to be run over the constructed graph applying the approaches for optimizing the solution and reporting the repercussions of this optimization comparing to the preliminary tests.

5.1 A time-dependent GIS dataset of the Netherlands

Demand for intelligent optimal routing has reached such a point that it has even triggered initiatives from relevant digital mapping and navigation systems enterprises to devise time-dependency of road segments along with the modeled transportation network. Such an example is the speed profiles dataset from Tele Atlas,[1] which covers most of United States, Canada and Europe [7]. It has been generated using data from probe, real driven speeds in the network (without considering maximum legal limits), after removing speed values below and above certain thresholds for minimum and maximum speed [6]. Speed profiles are 24h functions (Figure 5.1) with measurements at each 5 minutes of the day [6] that work as an 'impedance' (proportional) against measured freeflow speeds. These maximum speed values are average speed values during a period of least traffic, usually reflecting nighttime traffic flows [6]. Average speeds for weekdays and weekend days are also provided, which can be used in the absence of freeflow speed for topping up the profile speed values [6], as well as to use as constant-cost functions in the cases of network segments that were considered to be *non time-dependent* [7]. For each one of the 7 days of a week, a

[1]Tele Atlas BV and Tele Atlas North American

certain profile (or a default no profile value) is assigned to each segment of the transportation network through a link table (*HSNP Network Profile Link table*) that connects unique network identifiers in the vector dataset [5] with the speed profiles (Figure 5.2). Hence, it is only a matter of a lookup in the *HSNP* link table to know for a certain *network identifier* (or road segment) what is the speed profile that model the traffic behavior for a given day of the week. Once that is done, the speed profile data ((HSPR) permits to acknowledge the 'impedance' against the freeflow speed for a time slot of the day, and then lookup back to the *HSNP* link table to combine the 'impedance' values with the freeflow speed associated to the certain network identifier to derive speed values along the day.



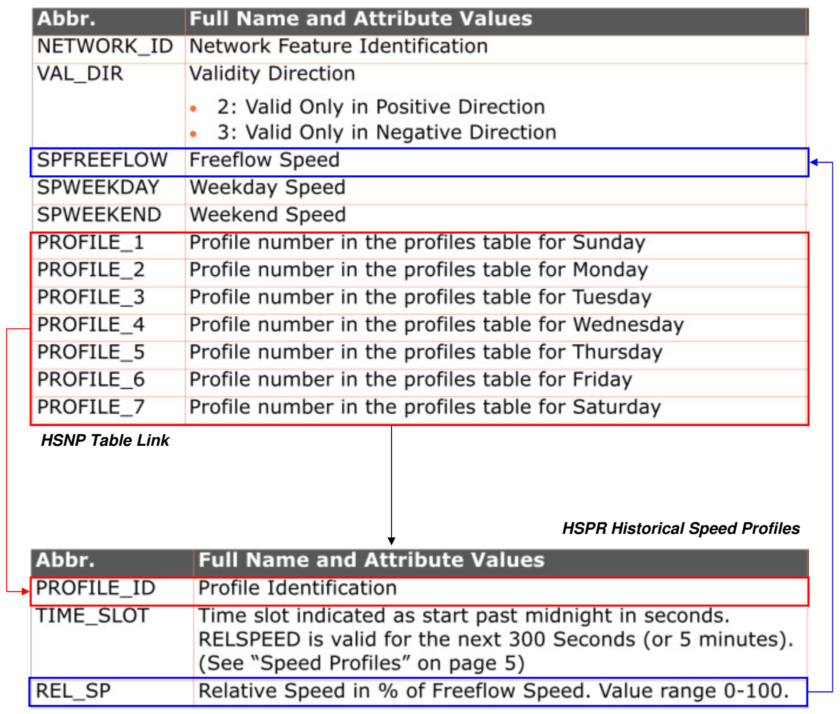
	profile_id integer	time_slot integer	rel_sp real
3769	14	52800	78.9
3770	14	53100	78.4
3771	14	53400	77.8
3772	14	53700	77.2
3773	14	54000	76.5
3774	14	54300	75.8
3775	14	54600	75
3776	14	54900	74.2
3777	14	55200	73.3
3778	14	55500	72.3
3779	14	55800	71.3
3780	14	56100	70.2
3781	14	56400	69
3782	14	56700	67.8
3783	14	57000	66.6
3784	14	57300	65.3
3785	14	57600	64
3786	14	57900	62.7
3787	14	58200	61.5
3788	14	58500	60.3
3789	14	58800	59.1
3790	14	59100	57.9
3791	14	59400	56.8
3792	14	59700	55.9
3793	14	60000	54.9
3794	14	60300	54.1

Figure 5.1: Snapshot of part of a speed profile data from Tele Atlas dataset.

The fact that the speed profiles are presented as relative values to the freeflow speed allows the same profile to be associated with more than one element of the transportation network that experience similar traffic conditions behavior [7]. In practice, this leads to the reuse of the profiles, and allowed Tele Atlas to reduce the representation of time-dependency in the network to only around 60 profiles per product tile (Figure 5.3) [7].

5.1.1 Characteristics of the transportation network

Tele Atlas dataset (Multinet® shapefile [5], Figure 5.4) for the Netherlands is composed of 1,198,395 transportation elements (i.e., segments in the network) that are open to traffic, totalizing 137,733.48 kilometers of roads in different levels of hierarchy (Table 5.1). From the total, around 16% belong to bidirectional transportation elements, i.e. road segments that are modeled as a single


HSNP Table Link
HSPR Historical Speed Profiles

Abbr.	Full Name and Attribute Values
NETWORK_ID	Network Feature Identification
VAL_DIR	Validity Direction • 2: Valid Only in Positive Direction • 3: Valid Only in Negative Direction
SPFREEFLOW	Freeflow Speed
SPWEEKDAY	Weekday Speed
SPWEEKEND	Weekend Speed
PROFILE_1	Profile number in the profiles table for Sunday
PROFILE_2	Profile number in the profiles table for Monday
PROFILE_3	Profile number in the profiles table for Tuesday
PROFILE_4	Profile number in the profiles table for Wednesday
PROFILE_5	Profile number in the profiles table for Thursday
PROFILE_6	Profile number in the profiles table for Friday
PROFILE_7	Profile number in the profiles table for Saturday

Abbr.	Full Name and Attribute Values
PROFILE_ID	Profile Identification
TIME_SLOT	Time slot indicated as start past midnight in seconds. RELSPEED is valid for the next 300 Seconds (or 5 minutes). (See "Speed Profiles" on page 5)
REL_SP	Relative Speed in % of Freeflow Speed. Value range 0-100.

Figure 5.2: Tele Atlas dataset link between speed profiles and transportation network segments (adapted from (6)).

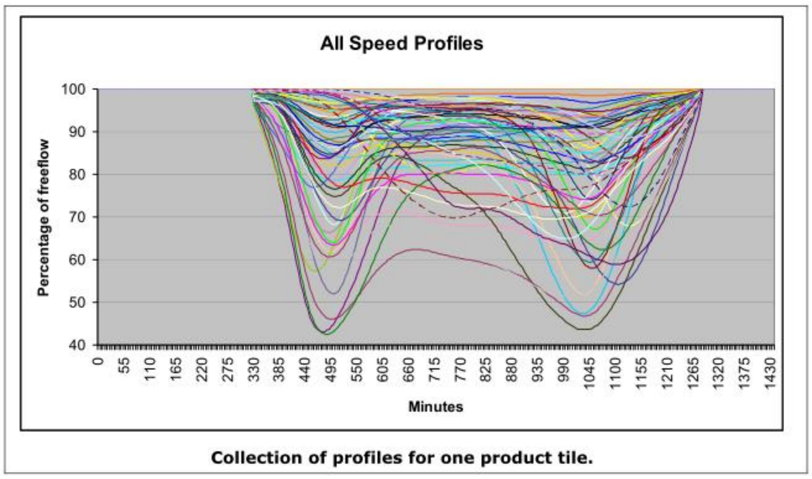


Figure 5.3: Polynomial fit of the 60 speed profiles for one product tile of the Tele Atlas dataset (7).

element in the network but actually allow flow in both directions; the remaining 64% are transportation elements that only allow traffic flow in one direction. Unidirectional elements are most common in the higher hierarchy levels of transportation, such as motorways and other main roads (Table 5.1). Apart from the network segments, which are modeled as linestrings, the dataset also presents a set of 1,003,346 nodes (modeled as points) that are connected by the network segments.

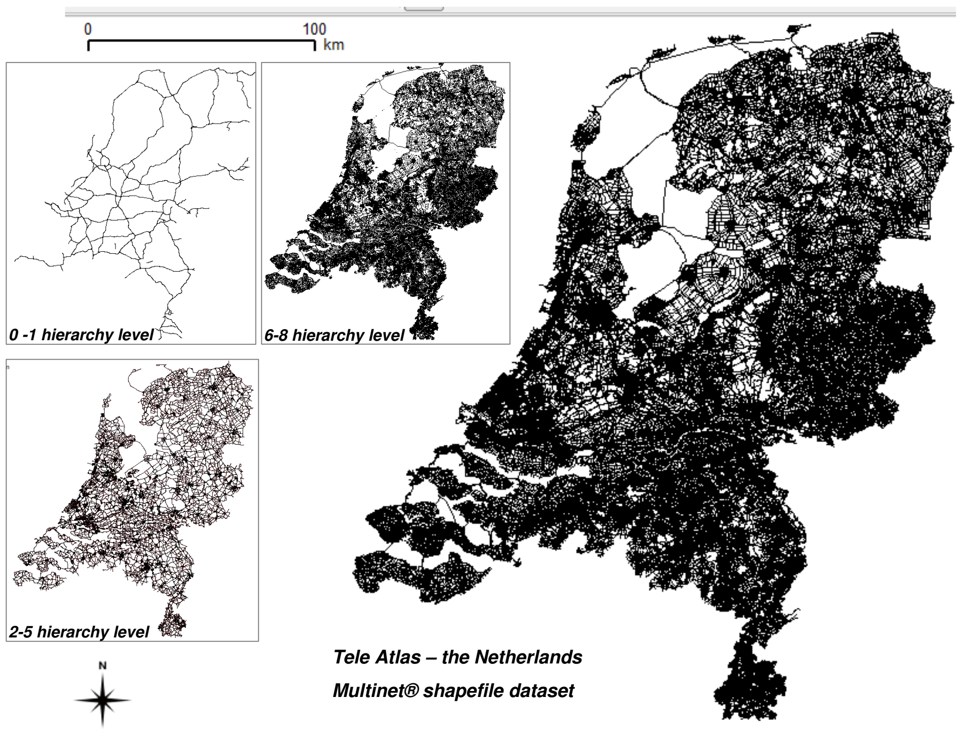


Figure 5.4: Tele Atlas transportation network dataset

As one may expect, not all segments of a transportation network will exhibit significant changes in traffic conditions to be considered as a time-dependent road element. Indeed, for the Netherlands Tele Atlas dataset, any degree of time-dependency is only found in around one quarter of the total network (see Table 5.2 and Figure 5.5). The discussion whether this is enough to justify a time-dependent shortest path is somewhat out of the scope of this research. We are mainly interested that this is a setting believed to be more and more present in people's daily life: intelligent optimal routing devices that consider travel time dynamics in the network.

The original GIS dataset from Tele Atlas consists of a set of edges (or linestrings) that connect nodes (or points) with each other, composing already a sort of graph representation. Some issues need to be considered to properly abstract the available network in a graph, so that the developed time-dependent solutions may perform unto it. In the following Section (5.2), we focus on how to transform the GIS dataset of the Netherlands into graph-theoretic dialect.

Table 5.1: Distribution of direction of modeled segment by hierarchy in the Tele Atlas dataset roads network.

Hierarchy level	**Direction**	**Proportion**
0: Motorway, Freeway, or Other Major Road	Unidirectional	3.87%
	Bidirectional	0.00%
	(Total)	3.87%
1: Major Road Less Important	Unidirectional	0.25%
	Bidirectional	0.11%
	(Total)	0.36%
2: Other Major Road	Unidirectional	2.37%
	Bidirectional	1.35%
	(Total)	3.72%
3: Secondary Road	Unidirectional	2.36%
	Bidirectional	3.12%
	(Total)	5.48%
4: Local Connecting Road	Unidirectional	2.35%
	Bidirectional	11.92%
	(Total)	14.27%
5: Local Road of High Importance	Unidirectional	0.98%
	Bidirectional	2.47%
	(Total)	3.45%
6: Local Road	Unidirectional	2.05%
	Bidirectional	30.28%
	(Total)	32.33%
7: Local Road of Minor Importance	Unidirectional	2.20%
	Bidirectional	34.30%
	(Total)	36.50%
8: Other Road	Unidirectional	0.00%
	Bidirectional	0.02%
	(Total)	0.02%

Table 5.2: Distribution of time-dependency in the Tele Atlas dataset roads network.

Category	**Km**	**Proportion**
Time-dependent (TD)	73,989.75	29.21%
TD but closed to traffic	119.73	0.05%
non TD and closed to traffic	536.54	0.21%
non TD and open to traffic	178,627.42	70.53%
TOTAL	253,273.4	100.00%

5.1.2 Loading the Netherlands time-dependent network dataset into a database

Tele Atlas road network dataset is basically composed of: (1) a vector dataset with the geometries of the roads network and their associated attributes (Figure 5.4); (2) a table with speed profiles (Figure 5.3); and (3) a link table that

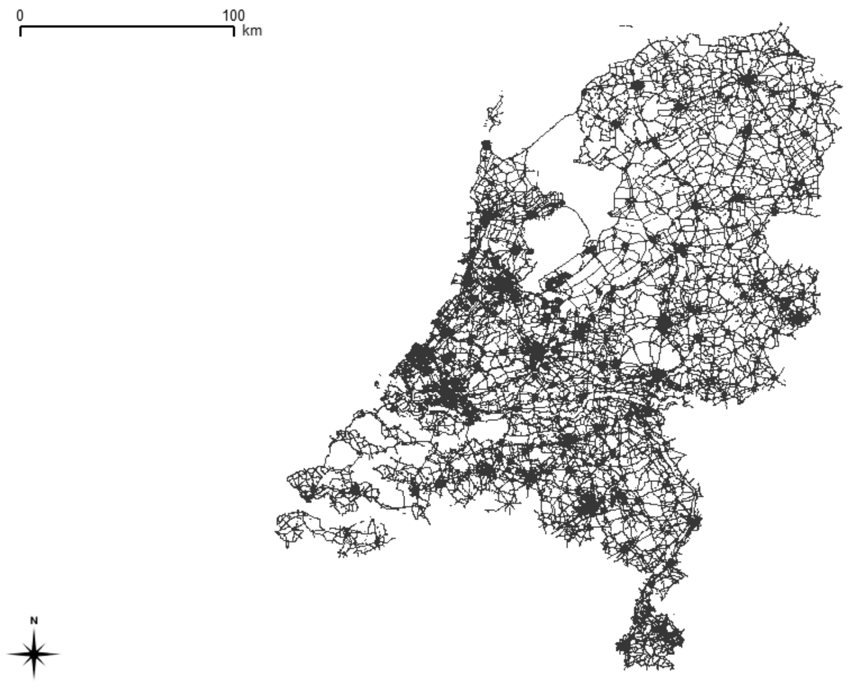


Figure 5.5: Time-dependent segments in the Tele Atlas dataset roads network.)

indicates which speed profile characterizes the traffic behavior of each day of the week in a given segment (*HSNP table*, Figure 5.2). The data was loaded into database especially created for the research project at ITC's PostgreSQL dataserver (`http://itcnt07.itc.nl`), following the steps summarized in Table 5.3. We refer to Appendix A for the source code preparing the dataset in the database.

5.2 Abstracting a Graph for applying (time-dependent) shortest path algorithms

Though a shortest path request in a transportation context can be seen as a spatial query, most of the solutions work with the abstraction of a graph to represent the spatial phenomenon of a transportation network. Thus, rather than overloading a shortest path algorithm with coordinates and geometries of objects, one can simply work with an abstract *graph* with its crucial characteristics allowing a graph-theoretic approach. Those characteristics can be summarized as:

Node identifier or what is being connected by the network? This is simply a unique identification of a node in the graph.

Table 5.3: Summary of steps carried out for loading the dataset into the database.

Dataset part	**Steps for loading**
Vector (geometries) network dataset	It is actually formed by two layers, one with the road segments and other with junctions (nodes that are connected by the segments). • First the original shapefiles with the network were converted to SQL insert commands by using the **shp2pgsql** data loader; • Then, the generated SQL insert commands were loaded into the database server using **psql** client terminal. None modification was made in the original dataset while loading into PostgreSQL server, apart from enforcing the name of the column to hold the geometry (*geometry_column*) in the destination table in the database to be named as *geom* instead of the default *the_geom* of **shp2pgsql** data loader.
Speed profiles data and HSNP link table	• The original tables (in dbf format) were converted to a text file tab delimited, while the structure of the tables were used as base for creating new empty tables in the PostgreSQL database; • Using **psql** client terminal, the data in the text file was loaded into the created new table in the PostgreSQL database with the command **copy**.

Edge identifier or what are the connectors in the network? Just like for nodes, edges must be uniquely identified in the graph. This identifier can also be used to associate an edge delay function with it.

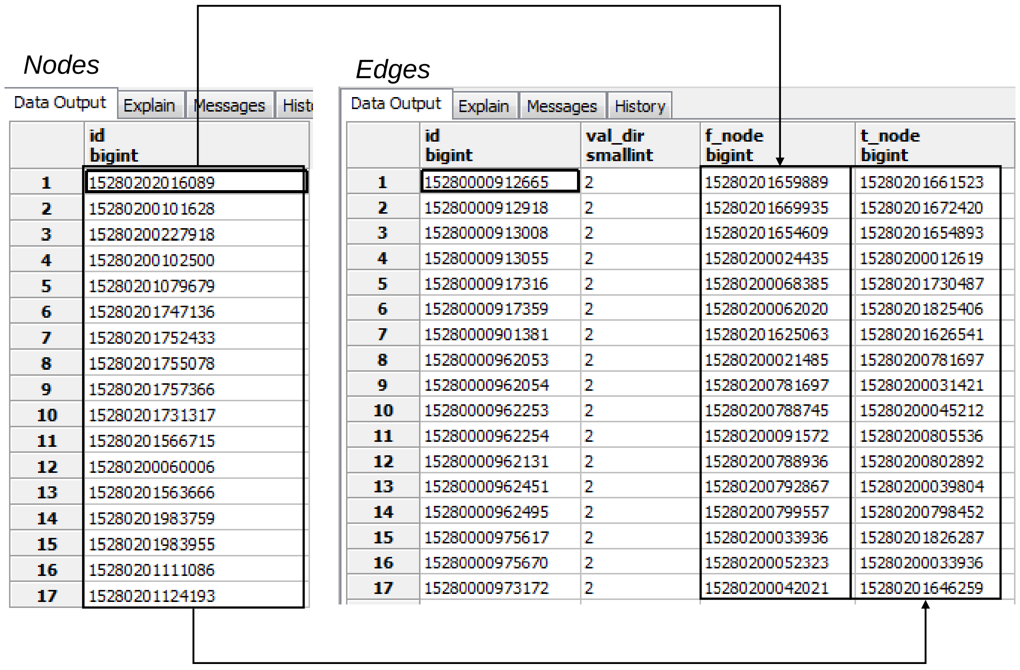
Edge connection or which edge is connecting which nodes? That is the essence of a graph: an edge connects a pair of nodes, unidirectionally or bidirectionally, therefore it has an origin (or *from node*) and a destination (or *to node*).

In the specific case of the Tele Atlas dataset roads network, bidirectional segments make use of an additional attribute[2] called *val_dir* (see Figure 5.2). This attribute identifies the validity direction to eventually distinguish speed

[2]In fact, any road segment presents such an attribute, but it is strictly needed only in cases of bidirectional segments.

profiles depending on the traversal direction. Our implementation assumes a directed graph, thus a bidirectional segment in the dataset is always replicated in the abstracted graph, with the exception that the direction validity attributes differ, and also the *from node* and *to node* are switched around. Figure 5.6 illustrates the table that abstracts the graph in the database for further use in the time-dependent shortest path request. A conceptual model of the graph plus the edge-delay functions that give the time-dependent character for the networks is further presented in Figure 5.7

The rather simple abstraction of the graph in the database allows to always go back to the original dataset, which includes the true spatial data. It also can easily be linked to speed profiles that characterize edge traffic patterns through the link table called *HSNP* (see Figure 5.2). Speed profiles are the basis for computing edge-delay functions which combine absolute time of the day (in seconds) with relative travel time to traverse a certain edge of the graph at that time of the day. This is determined at run time or can be precomputed and stored in the database.



Nodes

	id bigint
1	15280202016089
2	15280200101628
3	15280200227918
4	15280200102500
5	15280201079679
6	15280201747136
7	15280201752433
8	15280201755078
9	15280201757366
10	15280201731317
11	15280201566715
12	15280200060006
13	15280201563666
14	15280201983759
15	15280201983955
16	15280201111086
17	15280201124193

Edges

	id bigint	val_dir smallint	f_node bigint	t_node bigint
1	15280000912665	2	15280201659889	15280201661523
2	15280000912918	2	15280201669935	15280201672420
3	15280000913008	2	15280201654609	15280201654893
4	15280000913055	2	15280200024435	15280200012619
5	15280000917316	2	15280200068385	15280201730487
6	15280000917359	2	15280200062020	15280201825406
7	15280000901381	2	15280201625063	15280201626541
8	15280000962053	2	15280200021485	15280200781697
9	15280000962054	2	15280200781697	15280200031421
10	15280000962253	2	15280200788745	15280200045212
11	15280000962254	2	15280200091572	15280200805536
12	15280000962131	2	15280200788936	15280200802892
13	15280000962451	2	15280200792867	15280200039804
14	15280000962495	2	15280200799557	15280200798452
15	15280000975617	2	15280200033936	15280201826287
16	15280000975670	2	15280200052323	15280200033936
17	15280000973172	2	15280200042021	15280201646259

Figure 5.6: Abstraction of the graph in the database.

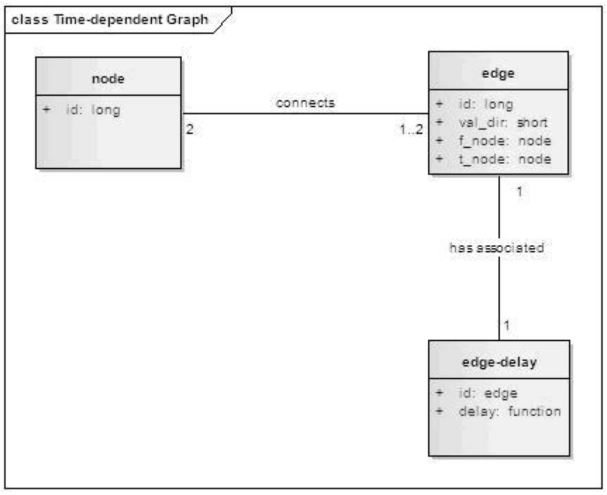


Figure 5.7: Conceptual model of the time-dependent graph.

5.3 Delivering edge-delay functions from speed profiles

A speed profile can be seen as a very efficient way of modeling traffic behavior of a road segment as it is independent of characteristics of the segment itself, particularly length and freeflow speed. Consequently, the same speed profile can be associated with more than one road segment to model the traffic behavior. On the other hand, speed profiles cannot be used straightaway for the purpose of finding shortest paths, as the road segment characteristics also play a role in defining what is shortest. In Figure 5.2, we have illustrated how true speed values can be derived from the speed profiles by considering the link table HSNP with its freeflow speed per road segment in the network. But this has to be further handle together with the segment distance to deliver edge-delay functions.

In our implementation, edge-delay functions are geometrically represented and stored in the database by using the data type Geometry, subtype LineString, with the use of PostGIS spatial database extension[3] for PostgreSQL. We start by transforming the speed profile data (pairs of time slot and relative speed) in 24h LineString functions for each of the 59 speed profiles presented in the Netherlands Tele Atlas dataset, but taking the *inverted relative speed* as y-value for the geometry (see Figure 5.8). In this form, the relative speed profile functions only need to be re-scaled by making use of the edge length and the freeflow speed associated to that (in HSNP link table) as illustrated in Figure 5.9. The illustration also points out that from the same speed profile different edge-delay functions may be derived when the characteristics of road segments are then taken into account.

[3]http://postgis.refractions.net/

```
**Algorithm** *invertedSpeedProfile*$(f(t))$
$pointarray \leftarrow union(serializePoints(f(t)));$
**for** $p \in pointarray$ **do**
    $p' \leftarrow makePoint(getX(p), 100/getY(p));$
    $update(pointarray, p');$
$f'(t) \leftarrow makeLineFromArray(pointarray);$
**return** $\{f'(t)\};$
```

Figure 5.8: Overview algorithm for getting profile with inverted speed values.

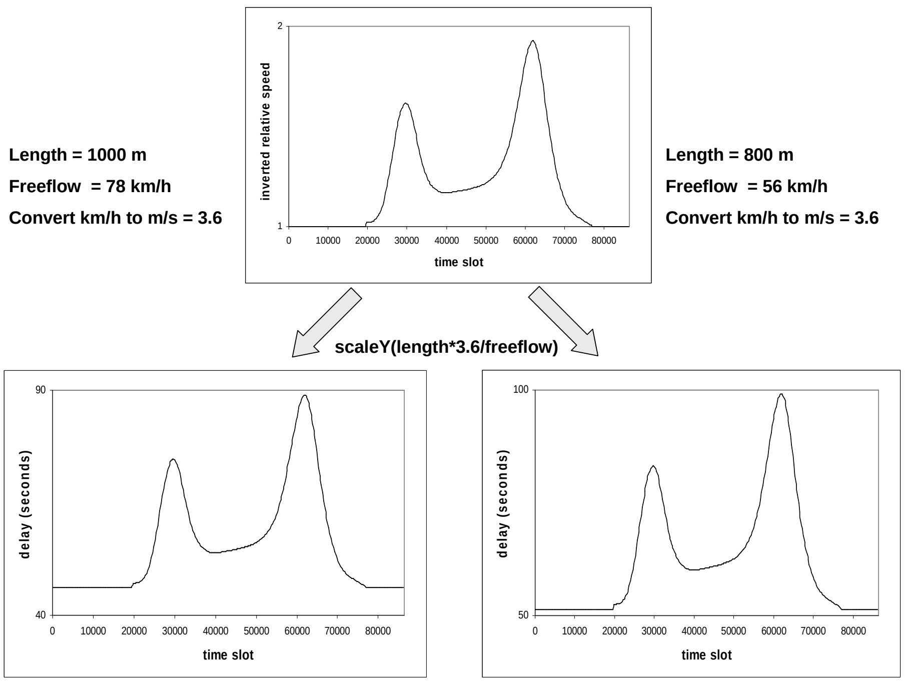


Figure 5.9: Derivation of edge-delay functions from inverted relative speed profile.

5.4 Solving TDSP-GDT in a database context with generalized Two-step LTT

We now present the implementation of the TDSP problem solutions described in Section 4.3 starting from the simple TDSP for a fixed departure time (TDSP-GDT). As our implementation is database-oriented, **SQL** is the preferred language to implement the functions in the database context, and we try to stick to it as much as possible. Eventually, the demand for procedural commands that are not presented in the plain SQL is fulfilled by using the procedural language of PostreSQL, called **PL/pgSQL**.[4] The latter can be seen essentially as

[4](http://www.postgresql.org/docs/8.4/static/plpgsql.html

a SQL-based language extended with procedural control flow such as assignments, loops and conditionals that strengthen the plain SQL in such a way that a broader nature of functions can be implemented.

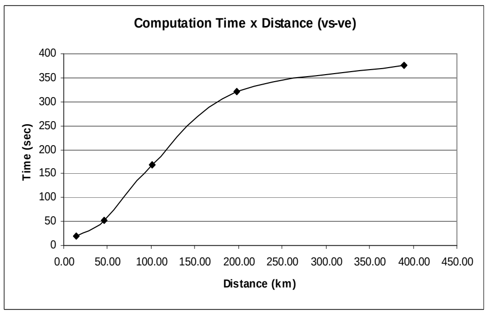
We refer to Appendix B for the source codes of the implemented tools. For the sake of clarity, we point out that we treat TDSP problems under the *non-overtaking property* (FIFO) assumption. That is a feature that arises from the fact that the dataset we use has the time-dependency modeled under a *flow speed* paradigm rather than directly in intervals of travel times. Sung et al. (2000) [83] shows that in such a way the FIFO assumption can be satisfied when computing the arrival-time functions. When that holds the principle of optimality is also satisfied and therefore any shortest path algorithm based on label-setting or label-correcting like Dijkstra's can be used [51]. Furthermore, we also clarify that in our implementation we assume that the travel is always performed completely within a day, therefore only the edge-delay functions of the given day are necessary to compute a time-dependent shortest path.

5.4.1 Preliminary computational results: towards fast computation

To get a first impression of performance of our solution for TDSP-GDT problems, we make use of a graph that contains only the motorways and major roads in the Netherlands (hierarchy level 0 and 1 in the Tele Atlas dataset; see Figure 5.4). This graph is composed of 18,263 edges and 16,947 nodes, and represents the "backbone" of intercity road transportation across the Netherlands. We apply the implemented solution for a set of requests arbitrarily chosen with increasing distance between source node v_s and target node v_e, for a given departure time arbitrarily defined as 30,600 seconds into the day (also known as 08:30 am). Figure 5.10 summarizes the characteristics of these first tests requests along with the computational results in terms of runtime complexity and number of nodes visited in the graph. These requests were tested before creating any index on tables in the database, and with the edge-delay functions for the graph being computed at runtime in the initialization phase of the algorithm. Results suggest that on average the shortest path computation performs in a rate of 1 second per km of path. The shape of the curves of computational results (Figure 5.10) demonstrates a clear reduction on the steepness as the number of visited nodes approaches the total number in the graph. This only indicates the shrinking of the search's possibilities as in many directions it will have encountered a 'dead end' (i.e., it cannot go further in the graph on that direction).

An attempt to improve the response time for the TDSP-GDT requests can be done by means of indexing tables in the database that are constantly used in the algorithm code (see Appendix A), as well as by precomputing the edge-delay functions before applying the TDSP-GDT solution to the graph. Results show that considerable improvement in computation time mainly occurs only for shorter distances of requests, which suggests that the optimized parts of the code, either by indexing of column or by precomputing edge-delay functions, were most likely not responsible for much of the time to compute the shortest

vs	15280200476341	time (ms)	19680
ve	15280200592440	distance(km)	14.49
ts	30600	nodes visited	163
dw	4	travel-time(sec)	568
vs	15280200802144	time (ms)	52036
ve	15280200027940	distance(km)	46.60
ts	30600	nodes visited	1270
dw	4	travel-time(sec)	1822
vs	15280200802144	time (ms)	168259
ve	15280200027940	distance(km)	101.32
ts	30600	nodes visited	5432
dw	4	travel-time(sec)	3576
vs	15280200016375	time (ms)	321105
ve	15280200021905	distance(km)	198.10
ts	30600	nodes visited	12341
dw	4	travel-time(sec)	6915
vs	15280201465367	time (ms)	375485
ve	15280200021905	distance(km)	389.95
ts	30600	nodes visited	15324
dw	4	travel-time(sec)	13933



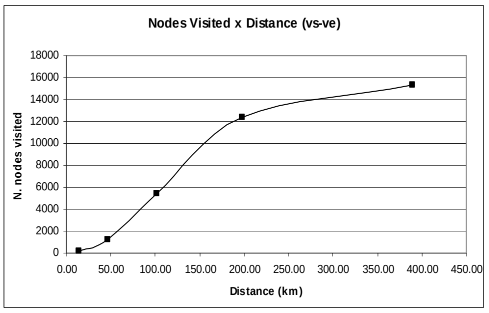
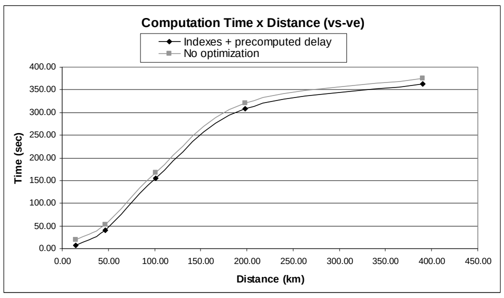


Figure 5.10: First impression on performance for TDSP-GDT solution in a database context for the case study.

path (Figure 5.11). What's more, the apparent considerable decrease in computation time for request with less distance $v_s - v_e$ is perhaps mainly due to the eradication of the step of computing edge-delay functions, which is a rather constant gain in computation time no matter the size of the request. Therefore, its effect starts to get diluted in the overall runtime as the size of the request increases.

These preliminary results can or can not be considered to provide satisfactory performance, depending on the context one would like to consider such a database-based solution to be applied. For instance, if we consider a scenario of en route navigation, with the user posing requests and wanting to get responses "on the fly," one may ponder that up to six minutes of computation time is not a desirable delay. On the other hand, if we consider a scenario in which a logistics and transportation enterprise wants to define the routes for their vehicles on a planning basis, those six minutes of computation may eventually be acceptable. All in all, we mean that every performance test may far differ in acceptability according to the context of application. But that only triggers our willing for finding better ways of handling the problem to get as faster results as possible. We present a way of doing so by applying a simplification to the graph, as it is presented in the following.

vs	15280200476341	time (ms)	7378
ve	15280200592440	distance(km)	14.49
ts	30600	nodes visited	163
dw	4	travel-time(sec)	568
			1
vs	15280200802144	time (ms)	39715
ve	15280200027940	distance(km)	46.60
ts	30600	nodes visited	1270
dw	4	travel-time(sec)	1822
			1
vs	15280200802144	time (ms)	155175
ve	15280200027940	distance(km)	101.32
ts	30600	nodes visited	5432
dw	4	travel-time(sec)	3576
			1
vs	15280200016375	time (ms)	307655
ve	15280200021905	distance(km)	198.10
ts	30600	nodes visited	12341
dw	4	travel-time(sec)	6915
			1
vs	15280201465367	time (ms)	362947
ve	15280200021905	distance(km)	389.95
ts	30600	nodes visited	15324
dw	4	travel-time(sec)	13933



Factor of decrease when applying indexes and precomputing edge-delay functions:

decrease factor
-62.51%
-23.68%
-7.78%
-4.19%
-3.34%

Figure 5.11: First impression on performance for TDSP-GDT solution in a database context for the case study, after creating indexes to constantly used table columns as well as precomputing edge-delay functions to the graph.

5.4.2 Graph simplification—or how to get faster TDSP-GDT problem results

One adverse characteristic of Tele Atlas dataset is the high degree of segmentation of the roads. This is understandable on the perspective of reuse of speed profiles. Perhaps, that is why Tele Atlas could reduce to around 60 profiles per product tile. The disadvantage of this is, however, that it is unnecessarily time consuming to the TDSP-GDT algorithm, as it needs to visit nodes in the graph that will eventually only lead to a single possible way. These edges could actually be collapsed into a single one in the graph (Figure 5.12). The first trivial simplification of the graph can be described as follows:

- We identify the nodes of V that are truly necessary in the graph by applying the condition that $\{v \in V \mid [[e \in E \,|e.f\_node = v.id \vee e.t\_node = v.id \bullet count(v.id)]] = 1 \vee [[e \in E \,|e.f\_node = v.id \vee e.t\_node = v.id \bullet count(v.id)]] > 2\}$ and then to generate the set of nodes $S \subset V$ that are either in the borders of the graph ($count(v.id) = 1$) or in junctions of real intersection of edges (i.e., junctions that have more than one possible way out, or more formal $count(v.id) > 2$).

- To “collapse” the abstracted graph by restarting from a given node in the set V of truly necessary ones and gradually collecting edges until the search finds another node in the same set (see Figure 5.13).

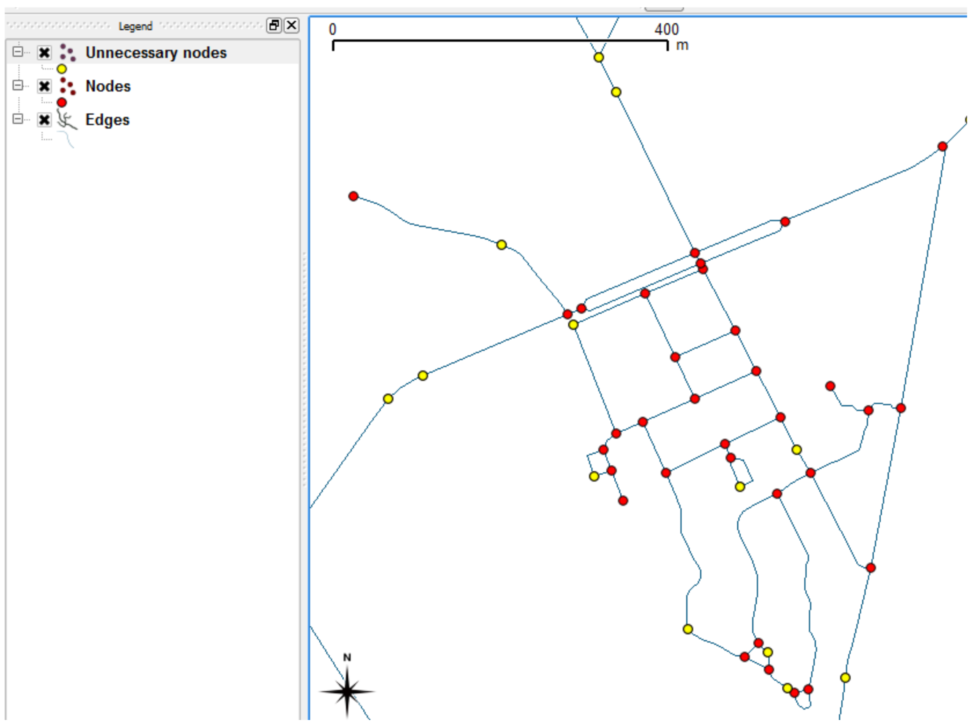


Figure 5.12: A snapshot with an impression on unnecessary nodes in the graph that could be “dissolved”.

We use this simplification without any change in the geometries of the road network, with all the modifications occurring at the level of the abstracted graph. This allows to always refer back to the original graph, as well as to the original geometries in the network. At this point, the intention is to prove the concept of optimizing computation time by simplifying the graph, we only apply it to a subset of the graph of motorways and major roads, i.e. hierarchy levels 0 and 1 in the dataset (Figure 5.14). Similar approaches are found in the literature for constructing a *contracted highway network* in hierarchical techniques to speed up computation of shortest paths (e.g., [76]), and as introduced in Section 3.3. To impose a restriction on the minimum degree of the nodes as two is found in the literature as identifying the 2-core of a graph as well [23].

The remaining difficulty in this context is to determine aggregated edge-delay functions that are originally apart when collapsing a set of edges into a single one. This has to be done by incrementally summing-up the edge-delay values of the segments but also considering the time passed since one departed

```
**Algorithm** *dissolveEdges*$(G_T(V,E,W))$
$newid \leftarrow 0;$
$S \leftarrow \{v \in V \mid [[e \in E \mid e.f\_node = v.id \vee e.t\_node = v.id \bullet count(v.id)]] = 1 \vee$
$[[e \in E \mid e.f\_node = v.id \vee e.t\_node = v.id \bullet count(v.id)]] > 2\};$
for $\{(v_i,v_j) \in E \mid v_i \in S\}$ do
    $newid \leftarrow newid + 1;$
    $update(D,(newid,(v_i,v_j)));$
    while $v_j \notin S$ do
        $v_i \leftarrow v_j;$
        $update(D,(newid,(v_i,v_j)));$
return $(S,D);$
```

Figure 5.13: Dissolve edges algorithm.

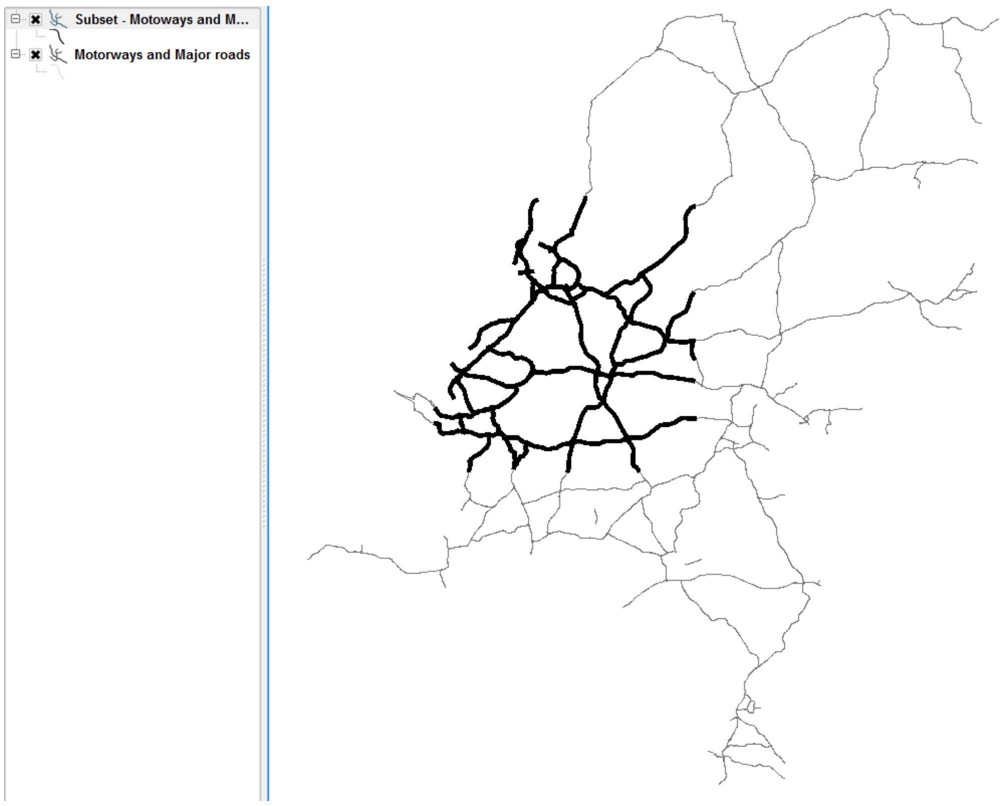


Figure 5.14: Subset of the graph to which the simplification of the graph was applied.

from a node and is traversing the segment (i.e., formally $w_{i,j}(t+w_{jk}(t+w_{i,j}(t))+\ldots$; as in the relevant concept introduced in Section 4.2. Figure 5.15) shows the overview of the algorithm that performs this aggregation of edge-delay functions.

Getting rid of unnecessary nodes in the graph is an effective way of reducing computation time, as a set of performance tests shows (Figure 5.16). Consider-

```
**Algorithm** *addEdge-delayFunctions*$(G_T(V, E, W))$
$(S, D) \leftarrow dissolveEdges(G_T(V, E, W))$;
Let $L$ be a list of pairs $newid \in D$ and aggregated edge-delay functions $\{w'_{i,j}(t) \,|i, j \in S\}$;
for each $newid \in D$ do
    $w'_{i,j}(t) \leftarrow w_{i,j}(t)$;
    if $\|newid \in D\| > 1$ then
        while $v_j \notin S$ do
            $v_i \leftarrow v_j$;
            $w'_{i,j}(t) \leftarrow w'_{i,j}(t) + w_{i,j}(t + w'_{i,j}(t))$;
    $update(L, (newid \in D, w'_{i,j}(t)))$;
return $L$;
```

Figure 5.15: Sum-up edge-delay functions of simplified edges algorithm.

able gain in computation can be clearly noticed; for instance, while a request with distance between origin and destination ($v_s - v_e$) of about 100 km takes around 150 seconds to be computed after applying indexes and precomputing edge-delay functions, in the simplified graph that would only take around 1 second. Unfortunately, this simplification has somewhat limited use in the case study dataset as the hierarchy of roads decreases towards local roads, meaning that the degree of segmentation of the network reduces in this direction. Table 5.4 summarizes how much reduction in graph size (number of nodes) can be obtained by the simplification as more hierarchy levels are aggregated to the graph.

Table 5.4: Reduction in the number of nodes by applying the simplification of graph with gradual inclusion of other levels of hierarchy levels of roads.

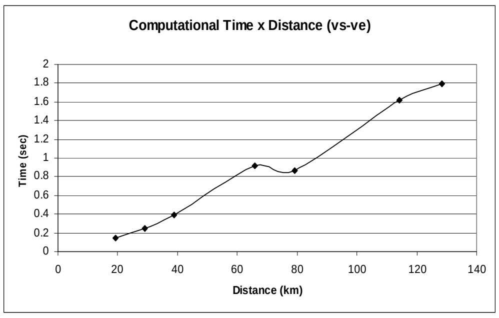
Road class (Hierarchy)	**Original**	**Simplified**	**Rate**
Motorway + Major (0-1)	16947	1510	91.09%
+ Other Major + Secondary (2-3)	96533	24258	74.87%
+ Local (4-6)	557887	204161	63.40%
+ Local or Minor Importance + Others (7-8)	939830	683086	27.32%

The speed-up achieved by applying this trivial simplification to the graph in our case followed roughly the shrinking in the size of the graph. Thus, by reducing around 90% the graph size the runtime results went down more or less in the same proportion.

5.5 Solving TDSP-LTT in a database context with Two-step LTT

We now move towards the more complex TDSP problem of our research project, in which a time interval for optimizing the departure time is given in the request. In the Two-step LTT approach the solution first determine what is the

vs	15280200603080	time (ms)	140
ve	15280200060167	distance(km)	19.15
ts	32400	nodes visited	10
dw	4	travel-time(sec)	746
vs	15280200617007	time (ms)	250
ve	15280200056644	distance(km)	28.96
ts	32400	nodes visited	26
dw	4	travel-time(sec)	1119
vs	15280201761173	time (ms)	390
ve	15280200097175	distance(km)	38.83
ts	32400	nodes visited	79
dw	4	travel-time(sec)	1588
vs	15280200113080	time (ms)	919
ve	15280200031354	distance(km)	65.87
ts	32400	nodes visited	194
dw	4	travel-time(sec)	2652
vs	15280200476341	time (ms)	865
ve	15280200045409	distance(km)	79.31
ts	32400	nodes visited	202
dw	4	travel-time(sec)	3083
vs	15280200886578	time (ms)	1621
ve	15280200022393	distance(km)	114.06
ts	32400	nodes visited	390
dw	4	travel-time(sec)	4187
vs	15280200894008	time (ms)	1795
ve	15280201840995	distance(km)	128.44
ts	32400	nodes visited	418
dw	4	travel-time(sec)	4784.63



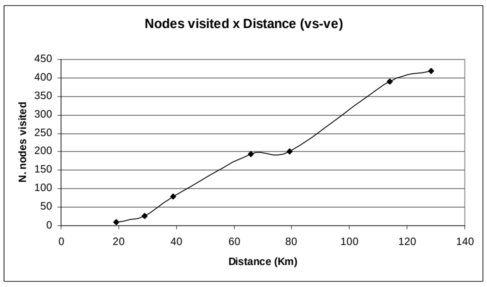


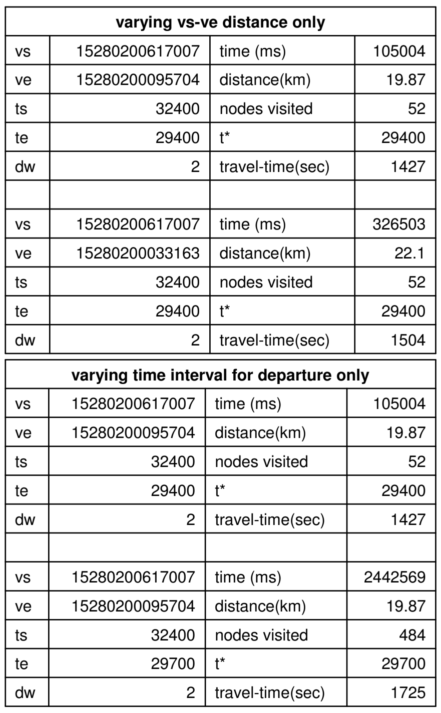
Figure 5.16: First impression of speeding-up the performance of TDSP-GDT solution in a database context for the case study, after simplifying the graph and generating aggregate edge-delay functions.

optimal time t^* of departure within the given interval of request by minimizing the total travel time (LTT) between source and destination and then fetch the path that leads to the computed LTT. We once again refer to the Appendix B for the source code of the functions implemented.

5.5.1 Preliminary computational results: finding limitations

According to Ding et al. [28] the time complexity of the Two-step-LTT algorithm is defined as $O((n \log n + m)\alpha(T))$ for n nodes and m edges and m cost functions $w_{i,j}(t)$, with $\alpha(T)$ designating the manipulation a function operation in time interval T. In other words, the complexity of the computation is dependent not only on the size of the graph (i.e., number of edges and nodes) but also on handling the time-dependent functions.

Aware of the higher complexity of solving TDSP-LTT, we start to get an impression of computational performance of the algorithm by applying it straightaway to the graph after the trivial simplification (see Section 5.4.2) has been achieved. The first impression of computational results was not very convincing, as the algorithm only finished processing in a reasonable time for rather small request sizes in terms of both distance between source v_s and target node v_e, and interval for departure time T (Figure 5.17). Basically, two factors are believed to contribute to this situation:



varying vs-ve distance only			
vs	15280200617007	time (ms)	105004
ve	15280200095704	distance(km)	19.87
ts	32400	nodes visited	52
te	29400	t*	29400
dw	2	travel-time(sec)	1427
vs	15280200617007	time (ms)	326503
ve	15280200033163	distance(km)	22.1
ts	32400	nodes visited	52
te	29400	t*	29400
dw	2	travel-time(sec)	1504

varying time interval for departure only			
vs	15280200617007	time (ms)	105004
ve	15280200095704	distance(km)	19.87
ts	32400	nodes visited	52
te	29400	t*	29400
dw	2	travel-time(sec)	1427
vs	15280200617007	time (ms)	2442569
ve	15280200095704	distance(km)	19.87
ts	32400	nodes visited	484
te	29700	t*	29700
dw	2	travel-time(sec)	1725

Figure 5.17: First impression of the performance of TDSP-LTT solution in a database context for the case study, after simplifying the graph and generating aggregate edge-delay functions.

Slow growth of interval of time-refinement: Two-step LTT works by refining the arrival time in a subinterval, which should gradually grow until it refines the whole interval of search. In our case study, we noticed that this growth of interval eventually converges slow, particularly when the algorithm reaches a "cluster of nodes" close to each other, which in the case of the graph with only motorways and major roads represents junctions, with all possible entries/exits. This closeness of nodes affects the time expansion of parameter Δ of the Two-step LTT algorithm, which in turn leads to a small subinterval $I^{'}$ for refining the arrival-time functions at each iteration (see Figure 3.5), thus aversely affecting convergence.

Updating arrival-time function gets slower: As the problem evolves in the Two-step LTT algorithm, arrival-time functions start to get more complex as more points become concrete in their representation. That is a well-know issue on time-dependent routing in which the aggregation of two delay functions always have a worst-case of $P(f) + P(g)$, where P is the number of concrete points in the function [23]. This leads to a slowdown of the algorithm particularly in the update of arrival-time functions (Figure 5.18).

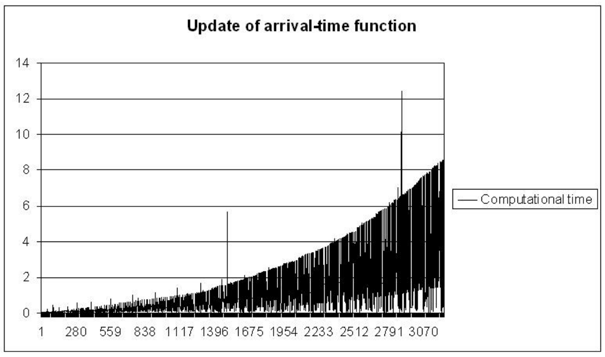


Figure 5.18: Computational time to update arrival-time functions as the iterations go on.

These results have shown that the algorithm demands solutions for reducing the complexity of the computation somehow, so that it can properly perform in large graphs and characteristics of dataset from real-world, as it is the case in our study. In the following sections, we explore possibilities toward this reduction of complexity for solving TDSP problems in a database context, using the chosen Two-step LTT approach.

5.6 Revisiting the graph simplification—or how to quickly traverse a dense part of the graph

The evolving of the subinterval to refine arrival-time functions in the Two-step LTT approach is dependent on the magnitude of edge delay values coming to a node in the graph, and therefore the growth of such subinterval slows down when the algorithm reaches a *dense subgraph* (a cluster of nodes). A way in which this problem could be overcome is by speeding-up the traverse of such a dense subgraph. In Section 5.4.2, we introduced the graph simplification in its trivial fashion, in which actually only unnecessary nodes in the graph are removed. We envisage another level of graph simplification by defining a sensible tile that encompasses that dense subgraph and substitute it by a *simplified subgraph* formed by the sets of *entry* and *exit nodes* and all the possible links to represent the possible combinations between entry and exit nodes (Figure 5.19). That will obviously lead to a reduction in the number of edges in the virtual graph down to the number of entry nodes times the exit nodes, but will also help by enlarging the size of the delays in the edges to be traversed, eventually speeding-up the growth of the subinterval of time refinement in the algorithm. The real method then is to generate aggregated edge-delay functions that properly represents the traverse of that original mesh of edges now substitute by the simplified subgraph. 'Aggregated' in this case is not a precise term; by aggregated case we actually mean to beforehand compute *least-delay functions* starting from each entry node and growing through the graph until it reaches each exit node.

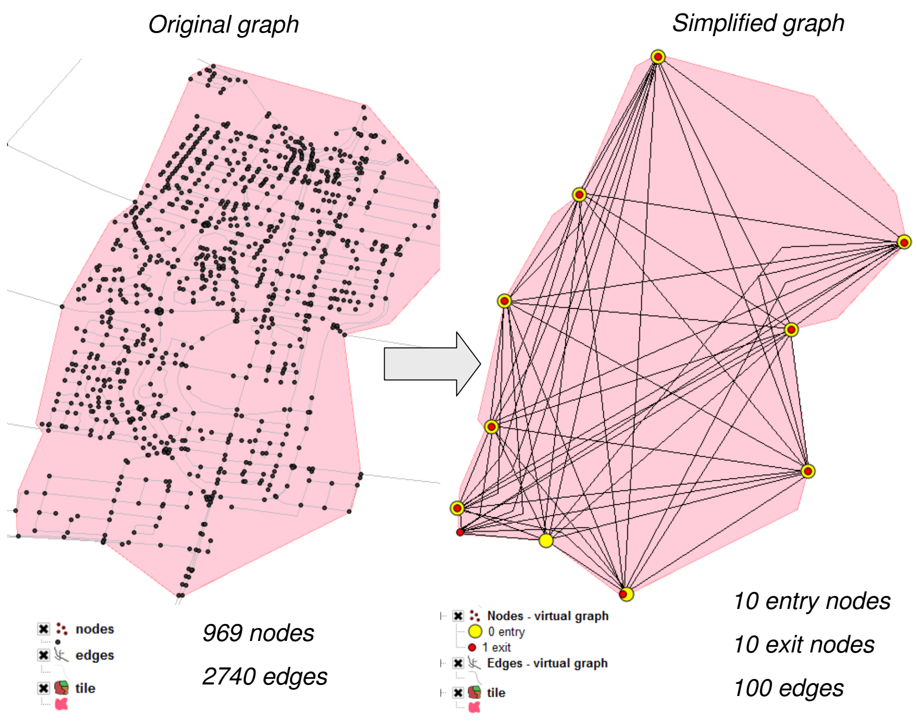


Figure 5.19: Intuition of the substitution by a simplified graph in a dense subgraph.

A conceptual model of the substitution of the graph in a certain dense subgraph by a simplified one is presented in Figure 5.20. In the model, "traverse_simplified_edges" class represents the substitution of the original 'mesh' of the graph by simplified edges that allow to traverse it at any possible combination of entry and exit node, while "inside_simplified_edges" is used to store least-delay functions to reach any intermediate node from one of the entry ones. The latter is not only necessary at running time, but also can be used to find the actual path through the original graph as will be further introduced.

Two challenges arise in the attempt of establishing the simplification. The first one is related to what is a sensible tile definition that encompasses a dense subgraph. Derived from that, the second is how to precompute the aggregated delay functions for the ins and outs of the defined tile. An overview of the simplification of a dense subgraph can be given as follows:

- Define a dense subgraph to be substituted by a simplified version.
- Determine the entry and exit nodes of the dense subgraph.
- Precompute least-delay functions that allow to traverse the dense subgraph from an entry to an exit node.
- Substitute the dense subgraph by the simplified version.

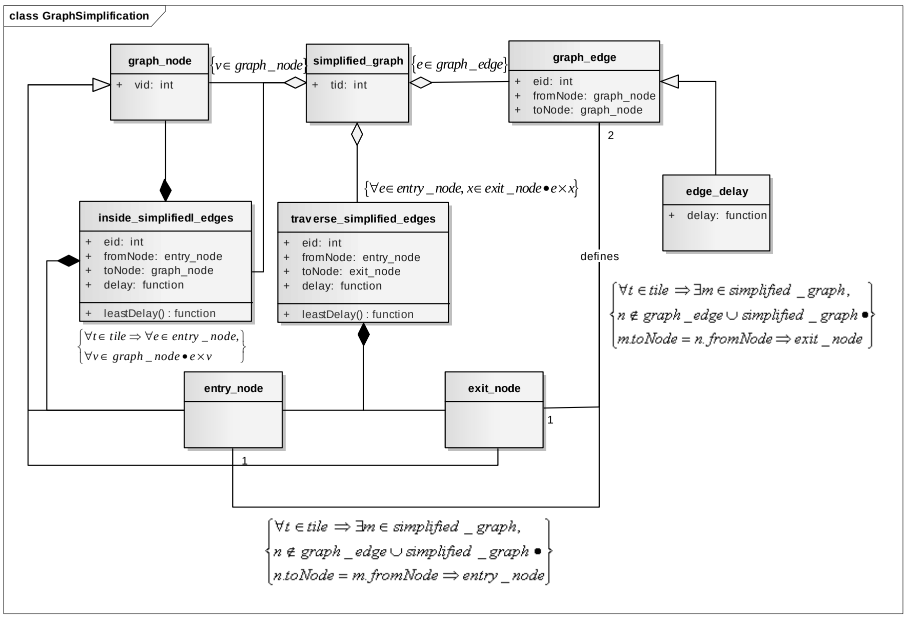

class GraphSimplification
graph_node
+ vid: int
{v∈ graph_node}
simplified_graph
+ tid: int
{e∈ graph_edge}
graph_edge
+ eid: int
+ fromNode: graph_node
+ toNode: graph_node
2
{∀e∈ entry_node, x∈ exit_node • e×x}
edge_delay
+ delay: function
inside_simplifiedl_edges
+ eid: int
+ fromNode: entry_node
+ toNode: graph_node
+ delay: function
+ leastDelay() : function
traverse_simplified_edges
+ eid: int
+ fromNode: entry_node
+ toNode: exit_node
+ delay: function
+ leastDelay() : function
defines
{∀t ∈ tile ⇒ ∃m ∈ simplified_graph, n ∉ graph_edge ∪ simplified_graph • m.toNode = n.fromNode ⇒ exit_node}
{∀t∈ tile ⇒ ∀e∈ entry_node, ∀v∈ graph_node • e×v}
entry_node
exit_node
1
1
{∀t ∈ tile ⇒ ∃m ∈ simplified_graph, n ∉ graph_edge ∪ simplified_graph • n.toNode = m.fromNode ⇒ entry_node}

Figure 5.20: Conceptual model of the substitution by a simplified graph in a dense subgraph.

We introduce the issues associated to this simplification of the graph in the following sections. An issue derived from that, which is the determination of the actual path through the dense subgraph that was simplified, is also discussed.

5.6.1 Defining a dense subgraph to substitute for the simplified graph

Any speed-up that relies on the definition of partitions of the graph may have its performance dependent on the method used for such task [60]. The kind of partitioning methods that we are particularly interested in are those that are based on density of the graph, or on the discovery of dense subgraphs. The latter has received quite good recent attention from research, for instance, in the context of web networks analysis [36] and bioinformatics [29]. Yiu & Mamoulis [88] propose a density-based and hierarchical variants of clustering techniques that apply distance in the network rather than Euclidean distance. Based on the *Single-Link* hierarchical clustering intuition presented in [88], we can formalize our definition of a dense subgraph of $G(V, E)$ by initially setting the number of cluster $k = \|V\|$, which means that each point is a cluster. Iteratively, clusters are merged based on their "closeness" until the number of clusters k is equal to H, where H is a tuning parameter that turns out to determine how dense the subgraph must be to consider it a cluster.

Another option is to first compute a road density layer for the graph and then define a threshold to separate the areas with dense subgraphs of interest. That would require the graph to have its layout, which is not a feature in the present way we abstract the graph. Nonetheless, as the original vector data is the basis for abstracting the graph, there is no difficulty in linking the layout (in the vector data) with the abstracted graph at all.

Yet another approach is to randomly iterate over the set of nodes in the graph solving one-to-all shortest path restricted by a tuning parameter H that would define how far a node can be to be considered in the same cluster. We present the algorithmic approach for this option in Figure 5.21. We assume that the graph given as input is a 2-core graph [23], i.e, a graph formed by a set of nodes with a minimal degree of two (see Section 5.4.2), as well as the existence of a function called *onetoallShortestPath* that takes as input the 2-core of a graph, a start node v_s and a parameter H which serves as stop criterion for the growth of the tree from the start node in network distance. The assumed *getTree* operator would be a mechanism to fetch the mesh of graph that was reached by the *onetoallShortestPath* search tuned by H.

```
**Algorithm** *clustering*$(G_T(V, E, W), H)$
Let $V_{2c}$ be a queue of the nodes $\in V$ in random order;
**while** $\|V_{2c}\| \geq 1$ **do**
    $v_i \leftarrow dequeue(V_{2c})$;
    $G_k \leftarrow getTree(onetoallShortestPath(G_T, v_i, H))$;
    **return next** $G_k$;
```

Figure 5.21: Define the dense subgraphs G_k.

5.6.2 Precomputing least-delay functions for the simplified graph

Computation of least-delay functions occurs inside a certain tile S that covers a mesh of the original graph G_T. We denote this graph mesh inside the tile as $G_k(V, E, W)$, and that is the input for the algorithm that performs such a computation (Figure 5.24). Apart from $G_k(V, E, W)$, the tile S also needs to have identified its *entry nodes* n and *exit nodes* x (Figure 5.22, Figure 5.23). *Entry nodes* are defined by finding a node of G_k to which there is no incoming edge in G_k. In contrast, *exit nodes* are defined by finding a node in G_k to which there is no outgoing edge in G_k. To compute least-delay functions it is cryptic to identify only the entry nodes in the subgraph in tile S, but exit nodes need to be identified to construct the simplified version of the subgraph as it is further introduced.

Operators $\oplus$ (Figure 5.25) and $\underline{min}$ (Figure 5.26) in the algorithm for computing least-delay functions for the dense subgraph inside a tile S are similar to the tools discussed in Section 4.4.3 and 4.4.4 respectively. An adaptation, though, is necessary for the former because the adding-up here works over two functions of t in which the y-values are both relative times (in the context of the Two-step LTT algorithm, this add-up is defined with one of the functions having y-values in absolute time). For both operators, the computation can be

```
**Algorithm** *entryNodes*$(G_k(V,E,W) \in G_T(V,E,W), G_T(V,E,W))$
$N \leftarrow \emptyset$;
**for each** $(v_i, v_j) \notin G_k(E)$ **and** $(v_j, v_k) \in G_k(E)$ **do**
    $enqueue(N, v_j)$;
**return** $N$;
```

Figure 5.22: Identify entry nodes of a tile S.

```
**Algorithm** *exitNodes*$(G_k(V,E,W) \in G_T(V,E,W), G_T(V,E,W))$
$X \leftarrow \emptyset$;
**for each** $(v_i, v_j) \in G_k(E)$ **and** $(v_j, v_k) \notin G_k(E)$ **do**
    $enqueue(X, v_j)$;
**return** $X$;
```

Figure 5.23: Identify exit nodes of a tile S.

```
**Algorithm** *leastDelay*$(G_k(V,E,W))$
$N \leftarrow entryNodes(G_k(V,E,W))$;
**for each** $v_j \notin N$ **do**
    $d_{nj}(t) \leftarrow \infty$;
**for each** $v_n \in N$ **do**
    $U \leftarrow \emptyset; C \leftarrow \emptyset$;
    $enqueue(U, v_n)$;
    **while** $U \neq \emptyset$ **do**
        $v_i \leftarrow dequeue(U)$;
        **for all** $(v_i, v_j) \in E$ **do**
            **if** $v_j \notin C$ **then**
                $enqueue(U, v_j)$;
        **for each** $(v_i, v_j) \in E$ **do**
            $d'_{n,j}(t) \leftarrow d_{n,i}(t) \oplus w_{i,j}(t + d_{n,j}(t))$;
            $d_{n,j}(t) \leftarrow \underline{min}\{d_{n,j}(t), d'_{s,j}(t)\}$;
            $enqueue(C, v_i)$;
**return**$\{d_{s,j}(t) \, | v_j \in V\}$;
```

Figure 5.24: Computing least-delay functions for the dense subgraph inside a tile S.

performed over the whole time interval T.

Precomputation of least-delay functions adds a requirement for representing the function in a desirable linear basis; that is, for a given dense subgraph S the need for extra storage after precomputing least-delay functions is linear w.r.t. to the size of the input graph in S. More precisely, let $N \subset V$ be the set of entry nodes in the dense subgraph S and let $X \subset V$ be the set of exit nodes in the same dense subgraph S. We call every node $s \in S$ such as that $s \notin N \bigcup X$, i.e., s is a node inside of the dense subgraph S that is not either an entry node nor an exit node. Thus, the precomputed least-delay functions

```
Algorithm addUpDelay($d_{n,i}(t), w_{i,j}(t), T$)
$pointarray \leftarrow union(serializePoints(d_{n,i}(t)), serializePoints(w_{i,j}(t))$;
for $p \in pointarray$ do
    $p' \leftarrow lookUp\_Yvalue(d_{n,i}(t), getX(p))+$
    $lookUp\_Yvalue(w_{i,j}(t), getX(p) + lookUp\_Yvalue(w_{i,j}(t), getX(p)))$;
    $update(pointarray, p')$;
$d'_{n,j}(t) \leftarrow makeLineFromArray(pointarray)$;
return $\{d'_{n,j}(t)\}$;
```

Figure 5.25: Add-up two edge-delay functions algorithm.

```
Algorithm minimalDelay($d_{n,j}(t), d'_{n,j}(t), T$)
$pointarray \leftarrow union(serializePoints(d_{n,j}(t), serializePoints(d'_{n,j}(t))$;
for $p \in pointarray$ do
    $p' \leftarrow makePoint(getX(p), mind_{n,j}(getX(p)), d'_{n,j}(getX(p)))$;
    $update(pointarray, p')$;
$d_{n,j}(t) \leftarrow makeLineFromArray(pointarray)$;
return $\{d_{n,j}(t)\}$;
```

Figure 5.26: Getting the minimal of two functions tool algorithm.

require $O(n(s+x)\alpha T)$, where αT designates the space to store a delay function of S, as extra space storage in the database.

5.6.3 Finding the actual path through a dense subgraph from least-delay functions

When a dense subgraph defined as a tile is substituted for a simplified version, the whole idea behind it is to speed up the time refinement step of finding a TDSP using the Two-step LTT approach. Still, the actual path through that dense subgraph has to be eventually found, and that demands a slight modification of the original algorithm for path selection. In the original approach, paths are found in a backward manner on the basis of the refined arrival-time functions in the first step of the algorithm, or more formally by meeting the following condition: $(g_i(t^*) + w_{i,j}(g_i(t^*)) = g_j(t^*)$. On the other hand, by substituting some parts of the graph for a simplified graph, not all node's arrival-time functions inside that mesh of the graph will be refined—in fact, it usually will be only the case for entry and exit nodes. That turns out to be infeasible to realize the path through that tile on the basis of arrival-time functions. But that can be easily adapted to use the least-delay functions that are previously computed and kept track of (see Section 5.6) for the graph mesh inside the tile. What is necessary for that is to have in mind that at moment the aggregated edge-delay functions start to be computed from a given entry node in the tile with a dense subgraph, it is actually defining a relative arrival-time function to the reaching node with relation to a certain entry node n in the dense subgraph. For the sake of distinction to the original approach, this aggregated edge-delay functions is what we call a *least-delay function* and we denote it as $d_{n,j}(t)$. Thus, the condition of selecting the predecessor node in the graph mesh inside the tile can be simply adapted to $(d_{n,i}(t^*) + w_{i,j}(t^* + d_{n,i}(t^*))) = d_{n,j}(t^*)$, and the path selection turns to be on the basis of the aggregated least-delay functions precomputed for the tile (see Section 5.6.2). From here storing a lest delay function $d_{n,j}(t)$ from any entry node n in the tile to any other node v_j in the mesh of graph inside of is important. Before searching for the path through a tile, it is necessary to acknowledge what is the simplified edge $v_n - v_x$ that is part of the TDSP solution, to then apply the $(d_{n,i}(t^*) + w_{i,j}(t^* + d_{n,i}(t^*))) = d_{n,j}(t^*)$, using the proper entry node as basis.

5.6.4 Substituting dense subgraphs by simplified subgraphs

The final step for allowing faster traversal of a dense subgraph is to actually perform the substitution of that part in the original graph for a simplified version with all possible combinations of entry and exit nodes. It is a matter of aggregating the previous steps (Section 5.6.1, 5.6.2) in a single procedure (Figure 5.27). That would allow to perform shortest path computation providing that the start and target node are not one of the nodes that were inside a simplified subgraph, i.e., to prove the concept of speed-up of computations by quickly traversing dense subgraphs that are simplified in the precomputation phase.

```
**Algorithm** *simplifyGraph*$(G_T(V, E, W), H)$
$G_k \leftarrow clustering(G_T, H)$;
**for each** $G_k$ **do**
    $d_{s,j}(t) \leftarrow leastDelay(G_k)$;
    $G'_T \leftarrow remove(G_T, G_k)$;
    $N \leftarrow entryNodes(G_k)$;
    $X \leftarrow exitNodes(G_k)$;
    **for each** $n \in N, x \in X$ **do**
        $update(G'_T, [n, x])$;
        $update(G'_T, createEdge(n, x))$;
        $update(G'_T, d_{n,x}(t) \in d_{s,j}(t))$;
**return** $G'_T$;
```

Figure 5.27: Substitution of the dense subgraph for a simplified version algorithm.

5.6.5 Proof-of-concept speed-up performance check

To prove the concept of substituting dense subgraphs by simplified versions that allow quick traversal of these graph mesh, we simplified the subset of motorways and major roads as shown in Figure 5.14. Though that subset was simplified in the trivial manner as introduced in Section 5.4.2, we apply the precomputation of least-delay functions (Section 5.6.2) for the original graph, hence getting an impression of performance that reflects a worst case. Nevertheless, when performing the substitution of dense subgraphs by simplified versions, the parts of the graph that are not substituted are integrated in the final graph with the trivial simplification. We refer to Appendix C for source code of the implementation of this proof-of-concept speed-up performance check.

The dense subgraphs to be substituted were arbitrarily defined in the absence of an implemented solution that would carefully delineate these subgraphs (see Figure 5.28). Inside the defined dense subgraphs there were 2,610 edges that turned out to become only 322 after simplifying the graph, which means a shrinking of around 88% in the dense subgraph size. The time taken to precompute all the steps carried out to substitute the dense subgraphs, plus fetching the rest of the trivially simplified graph that was not affected by the substitution is presented in Table 5.5. Overall precomputation time was almost 2 hours and a half, which cannot be said to be irrelevant, though worthwhile as it is a one time big computation for an expected considerable speed-up in solving the TDSP requests.

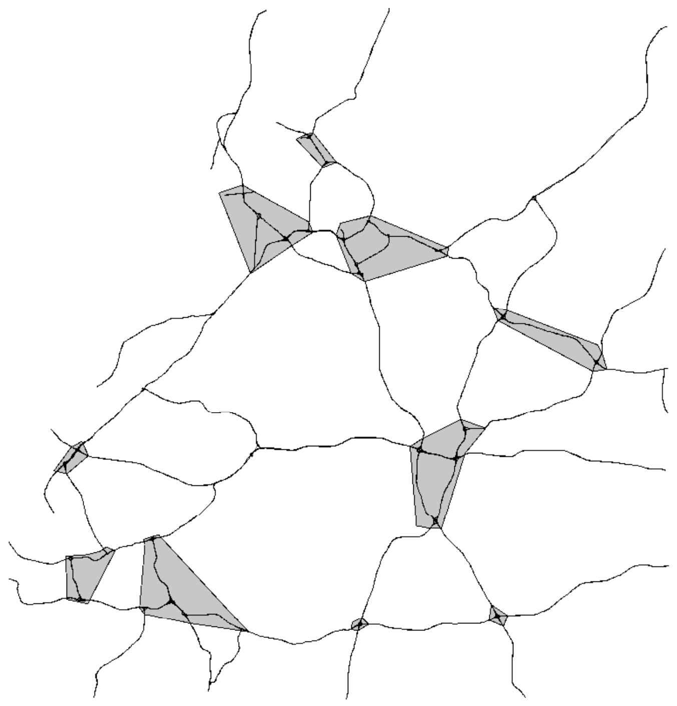


Figure 5.28: Arbitrarily defined dense subgraph to be simplified—first attempt.

The performance check results in applying the substitution of dense subgraphs by simplified versions are reported, only considering the time refinement of the Two-step LTT approach, as this is the step that can be sped up (and need to be, once it is by far the most expensive computation part of the algo-

Table 5.5: Precomputation time for substituting dense subgraphs by simplified version with least-delay functions to allow quick traversal of these dense subgraphs—first attempt.

Subgraph	**Computation time (ms)**	**(h)**
Cluster 1	364,843	0:06:05
Cluster 2	1,648,581	0:27:29
Cluster 3	124,875	0:02:05
Cluster 4	148,672	0:02:29
Cluster 5	1,629,232	0:27:09
Cluster 6	75,295	0:01:15
Cluster 7	1,090,894	0:18:11
Cluster 8	2,028,765	0:33:49
Cluster 9	1,537,537	0:25:38
Cluster 10	158,581	0:02:39
Rest of graph	1,716	0:00:02
TOTAL	8,808,991	2:26:49

rithm) using this simplification. We start by comparing the performance after the substitution of dense subgraphs by simplified versions against a graph with only the trivial simplification. It turns out that simplification to the clusters in Figure 5.28 considerably speed-up the computation of the time refinement of Two-step LTT but mainly when the interval of departure time to optimize is rather small such as in Table 5.8. The reason for that is probably the remaining of relatively small dense subgraphs in the simplified graph, leading once more to the slow growth of convergence of the refining interval. As a larger interval of departure time is given to refine, there is more opportunity for the search to reach farther nodes in the graph, eventually reaching one of the remaining dense subgraphs in the graph. By testing a gradual increase in interval for time departure for a fixed distance between start and end node of +30 km, it was found that up to 30 minutes of interval the problem can be solved in a several seconds based, just like as indicated in Table 5.8.

Table 5.6: Impression on speed-up achieved by the graph simplification in varying distance between start and end node (*interval for departure time refinement of 10 min).

Distance	**Before (ms)**	**After (ms)**	**Speed-up**
+30km	41,617	952	43.72
+50 km	649,277	4,883	132.97
+60 km	822,194	7,140	115.15
+70 km	898,141	8,018	112.02

To cope with the limitation of computation even after applying the simplification over the subgraphs in Figure 5.28, we redefine the dense subgraphs arbitrary delineation to avoid any occurrence of small delay values that would lead the solution to slowdown (Figure 5.29). That turned out to demand much more precomputation effort as show in Table 5.7.

We finally test the redefined dense subgraphs (Figure 5.29) with a relatively

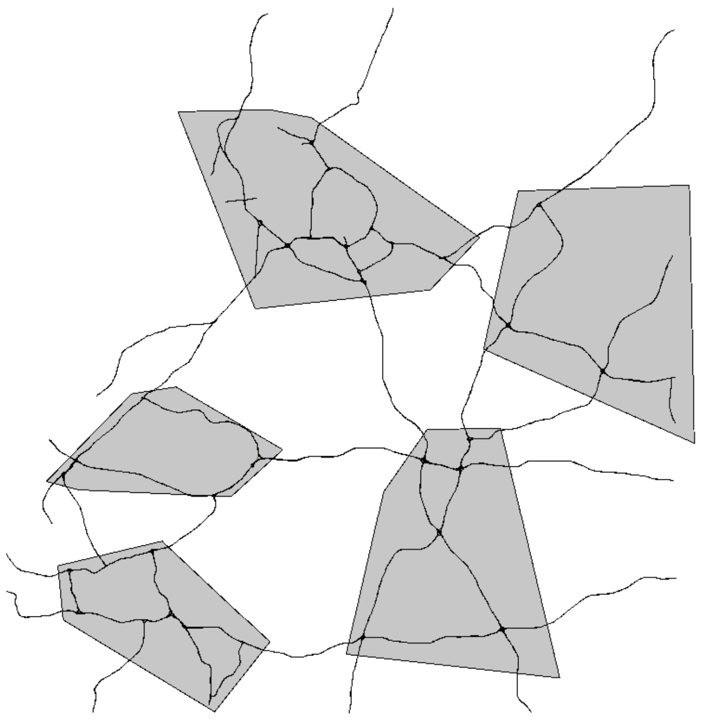


Figure 5.29: Arbitrarily defined dense subgraph to be simplified—second attempt.

Table 5.7: Precomputation time for substituting dense subgraphs by simplified versions with least-delay functions to allow quick traversal of these dense subgraphs—second attempt.

Subgraph	**Computation time (ms)**	**(h)**
Cluster 1	7,124,150	1:58:44
Cluster 2	2,165,601	0:36:06
Cluster 3	7,466,592	2:04:27
Cluster 4	2,451,464	0:40:51
Cluster 5	14,409,505	4:00:10
Rest of graph	1,185	0:00:01
TOTAL	33,618,497	9:20:18

large fixed distance between start and target node (+120 km) and gradually varying the interval for departure time refinement.

From the speed-up achieved by the second attempt of applying the substitution of dense subgraphs by simplified versions, it seems clear that one of the main shortcomes of the chosen approach can be overcome. We should point out, though, that our arbitrary definition of dense subgraphs to substituted was quite exaggerated, as we only wanted to prove our view that the simplified graph concept could cope with one of the limitations faced in the chosen algo-

Table 5.8: Impression on computational performance after graph simplification to allow quick traversal of dense subgraphs, with varying interval for departure time (*distance between start and target node of +120 km).

Interval for departure time	Computation time (ms)	(h)
30 min	88,226	0:01:28
45 min	201,226	0:03:21
60 min	358,213	0:05:58
120 min	1,607,564	0:26:48

rithm for solving TDSP-LTT problems. Nevertheless, the issue of slowdown was still prominent in the arrival-time functions update for a larger request size in terms of interval for departure time refinement as the problem evolves.

In terms of time complexity for the simplification of the graph, our results indicate a linear growth of runtime following the graph size inside the cluster (Figure 5.30). That is a feature which in general is desirable, as it would be prohibitive to apply the simplification in larger graphs if the precomputation complexity was polynomial in relation to the subgraph size.

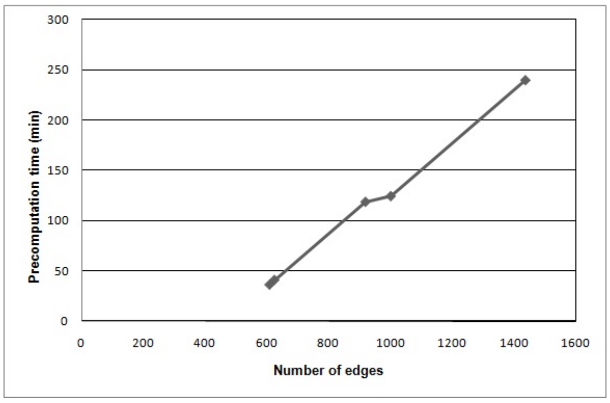


Figure 5.30: Runtime complexity in relation to the size of the subgraph when applying the graph simplification.

Similar relation was not found in terms of space complexity, which in our case particularly refers to the growth in number of concrete points in the edge-delay function after the graph simplification. But obviously the aggregation of edge-delay functions leads to an increase in space complexity. While in average the original graph mesh had maximum of 120 concrete points in the edge-delay functions, in the simplified version this number goes up to 193 points (i.e., an increase of 61%).

A final test was made in the attempt of getting an average runtime performance. We first analyze the average response in twenty randomly varying distances between start and target node, and fixed time intervals for departure (1h, 1:30h and 2h). Then we fix the distance between start and target node (+100 km), and randomly pick twenty time intervals for departure (within 10min and 2h). The results of the first average test are presented in Table 5.9, and show that considerable more runtime is needed to solve varying distance TDSP requests when the time interval for departure increases. The variation of the responses in relation to the average are indicated by the standard deviation, which is proportionally larger in smaller time intervals for departure. Which gives an indication that the varying distance primarily influences the runtime when the major computational expense contributor, i.e., *time interval for departure*, is not large enough to lead to more runtime. This is further demonstrated in the results of the second average test for randomly varying time interval for departure (within 10min and 2h) and fixed distance between start and target node. With an average of 9 minutes and 21 seconds, but with a standard deviation that is even larger than the average (0:09:35 h) for that test, the implemented solution suggest to be much more sensitive to larger time interval for departure than to increases in distance between start and target node.

Table 5.9: Average response over 20 requests with randomly varying distances between start and target node, and fixed time intervals for departure.

Interval for departure time	**Computation time (h)**	**Std. deviation (h)**
0:30h	0:01:08	0:00:35
1:00h	0:02:17	0:02:17
1:30h	0:15:36	0:05:44
2:00h	0:36:25	0:10:32

Our proof-of-concept performance check shows that considerably faster computation of TDSP can be achieved by use of some graph simplification technique. Nevertheless, the optimization solutions we proposed and developed are far from an exhaustive list of possibilities in this direction. We will briefly mention some others in the discussion of the following chapter.

Chapter 6

Discussion, Conclusions and Recommendations

In this chapter, we outline and present a discussion on the results achieved in this project, based on the research questions specified in Section 1.2.2, and discuss further the subject in its broader sense. That is followed by the conclusions drawn from the results obtained and the previous relevant discussion. Finally, we present the recommendations for future improvements or avenues that we believe are worthwhile of taking in follow-up research in the context of the present.

6.1 Outline of the results and discussion

Modeling the real-world dataset for applying graph-theoretic approach

The time-dependent network dataset we have used for our case study implementation was described in Section 5.1, including the main features related to the original data model. We refer to [6] for details on the way Tele Atlas has modeled the daily speed profiles, as representing the time-dependency of the network, and its relation to the spatial network. Some important characteristics of the dataset are summarized in Section 5.1.1, while Section 5.1.2 presents the transformation of the file-based dataset into a database system, with the sequence of codes used shown in Appendix A.

A next step was then to abstract a graph from the generated database system so that the developed graph-theoretic approach could operate on. This is outlined in Section 5.2, in which a layout-free abstraction was chosen, i.e., the graph only presents its topological relationships without involvement of geometries. This was followed by the description of generating edge-delay functions for the time-dependent part of the dataset which was originally presented as speed profiles (Section 5.3). At this point, we had two possibilities: (1) to generate (for the given day of the week) edge-delay functions "on-demand" together with an initialization phase every time the solution would be applied; or (2) to precompute (for all days of the week) edge-delay functions and only fetch (for the given day of the week) the needed edge-delay functions in the initialization phase. Obviously here there is a trade-off between additional runtime for option (1) and requirement for additional space for the precomputed edge-delay functions for option (2). Notice that in the latter we avoided to overload the TDSP solution with unnecessary data by fetching the needed precomputed edge-delay functions for the day of request. As a matter of fact, both options were tested along with a performance verification of the solutions developed in Section 5.4.

A transformation of the graph is presented in Section 5.4.2. It removes unnecessary nodes (i.e., nodes with degree two), and collapses the path in which these nodes lie into a single edge with aggregated edge-delay function. This is an important feature in optimizations of shortest path solutions, for instance, in the Highway Hierarchies [76, 77] approach introduced in Section 3.3. Moreover this transformation is part of tested measures to enhance the performance of the developed solutions discussed below.

Development of formulated TDSP problems solutions in a database context

In Section 4.1, two TDSP problems are formulated in the database context. The first problem formulation is a one-to-one time-dependent shortest path for a given departure time, called TDSP-GDT for short, in which the optimal path in travel time costs has to be found for a given start-of-travel moment. The second problem formulation is more complex: one-to-one time-dependent shortest path for a given time interval of departure, called TDSP-LTT, or time-dependent shortest path with least travel time. In which not only a path has to be found, but primarily the best moment for departure has to determined on the basis of the least travel time possible within the given departure time interval. This is followed by formally defining relevant concepts that are inherent to a solution of TDSP problems in Section 4.2.

A variant of Dijkstra's algorithm for static shortest path found in the literature [28], called *Two-step LTT*, is introduced in Section 4.3 for solving TDSP-LTT. The method is explained, also presenting examples to illustrate the algorithm. From the proposed method, we define what are the auxiliary tools needed to handle TDSP problems. We present the algorithmic approach for the development of the auxiliary tools in Section 4.4. The Two-step LTT algorithm is adapted for including the auxiliary tools as described in Section 4.5, in which the method is also generalized for TDSP-GDT.

Limitations on computational complexity performance

After implementing the chosen TDSP problem solutions in PostgreSQL/PostGIS, using the default procedural language of this DBMS, **PL/pgSQL** (Appendix B) a set of request tests was derived to get insight on the possible limitations on performance of the developed solutions. We started from the simpler TDSP-GDT and a subset of the original graph, with only the higher hierarchies of roads.

The results have shown that the algorithm is capable of finding a shortest path even for considerably large (up to around 400 km) distance between start and target note, though not with an impressive performance at first, when an average of 1 second of computational time per km of path was found. This first trial did not make use of any kind of optimization measure; for a second run of tests, indexes for frequently accessed columns in the database model as well with precomputation of edge-delay functions were included. In general, not much enhancement was achieved with these measures, and only considerable speed-up was noted for small request sizes (i.e., start and target node closer). This can be explained by a reduction of computation time in the initialization phase in which edge-delay functions were precomputed. As this rather small extra runtime for delivering edge-delay functions "on-demand" in the first tests is constant and independent on the size of the request (i.e., distance between start and target node), the obtained speed-up is only noticed in small runs, tending to get diluted over the heavier computational parts of the solution, as the problem solution evolves.

We then introduce a speed-up technique by collapsing connected edges that belong to a shortest path (actually, the only path possible) between two junction node in the roads networks. It is the realization that nodes with degree two are an unnecessary part of the graph, as they only have one way in and out, so the graph can be simplified. This same technique is used, for instance, in the speed-up technique for static shortest path known as Highway Hierarchies. By generating a single edge with an aggregated delay function the number of nodes in the test graph (higher hierarchy of roads) shrinks by more than 90%, and as a consequence runtime is reduced in the same order. Unfortunately, this trivial graph simplification has somewhat limited use, as the number of nodes in the graph reduces only by below 30% with the inclusion of lower hierarchies of roads.

If we can assume that the speed-up roughly follows the reduction in the graph size, one could expect a speed-up of around 30% when the simplification is applied to the complete graph set. Whether such a performance is good enough is a completely purpose-driven issue, and certain applications may well allow to more runtime. An example of the latter could be planning of routes in transportation and logistics realized in a time-dependent fashion—i.e., the costs are assumed to be known beforehand and there is no dynamic change of costs functions while the problem is being solved. If this planning routine is made with enough advance to the realization of the routes by the vehicle/fleet, then one may assume that the problem not necessarily need to be solved on blink-of-an-eye basis.

When moving to the more complex problem formulation of TDSP-LTT, however, the complexity of the task becomes much more apparent, even in a trivially simplified graph. A first run of tests obtained a not so impressive performance, and the problem could only be solved for rather small requests, either in distance between start and target node or in the size of the interval given for a departure time optimization.

An investigation to identify the most expensive computation parts of the algorithm defined a sensible division of the code and measured the performance of each part over several thousands of iterations. From this exercise, slight changes of the code were made to improve performance here and there, turning out that the culprit part of the code was found in the update of arrival-time functions (Section 5.5.1). This can be explained by the growth in data complexity (i.e., number of explicit points that define the function curve) as the algorithm proceeds. Delling [23] refers to this as one of the main issues in TDSP, where a composition of two functions $f \oplus g$ has as worst case of $P(f) + P(g)$, and P is the number of concrete points in the function.

Another issue identified as contributing to the long runtime was the slowdown in the growth of the subinterval (I') of time refinement when the problem evolving reaches a dense part of the graph in which the delay values are rather small. The subinterval is dependent on the Δ parameter, which is controlled by the magnitude of the least delay value coming to a node in iteration (see Figure 3.5), therefore when reaching a dense part of the graph, where the delay values are likely to be considerably smaller, Δ leads to a time refinement in only small steps approaching the end of the given interval for departure.

We then introduce a theoretical concept that can be used to speed up the computation using the proposed approach, by identifying dense subgraphs inside the original graph to be simplified, to allow fast traversal of that part of the graph. This fast traversal comes at cost of precomputation: (1) to determine the dense subgraphs; (2) to precompute 'aggregate edge-delay functions' that define *least-delay functions* to traverse a given dense subgraph[1]; and (3) to substitute the dense subgraph in the original graph for a simplified version. This is particularly helpful for traversing a part of the graph in which the target node does not lie, but can also be adapted for a situation in which the target node lies in the dense subgraph. The precomputation of least-delay functions between all combinations of entry and exit nodes in a dense subgraph namely ends up with least-delay functions to reach any node inside of the subgraph from any entry node of it. This also permits the determination of the actual path through a dense subgraph as described in Section 5.6.3.

A proof-of-concept test of the speed-up was obtained, showing that at the cost of a single large precomputation considerable speed-up can be achieved. But it also became clear that the definition of dense subgraphs must include even rather small dense subgraphs. If that does not hold, the computation can slow down when there is opportunity to the search of reaching a remaining small dense subgraph.

As ideally one should never make any sort of assumption on where the start and target nodes of a request are, it is clear that the situation in which one or both of them are inside a dense subgraph that was already simplified to allow fast *traversal* need to be carefully considered. One possibility consists of a fast mechanism that identifies the dense subgraphs in which neither start nor target node are and quickly substitute these parts by the precomputed simplified forms, while preserving the dense subgraphs where the these nodes are. Another possibility is a division of the problem in two levels, *local* and *global*, which seems to be in theory a good approach.

Local queries take care of the search where it is "close enough" to either start

[1]Assuming that the target node is not inside of the dense subgraph.

or target node.

Global query continues the search when "far enough" from the start node, and takes care of most of the long-distance travel speed-up by substitution of dense subgraphs for simplified versions until the search is "close enough" to the target node.

The concepts of "close enough" and "far enough" in this context are driven by the dense subgraphs substituted in the original graph. Therefore a *local query* delivers the search for the *global query* once it has solved the search until the possible ways out of the subgraph. Similarly, the *global query* returns the search back to a *local query* once the dense subgraph in which the target lies is reached. All of this should be devised in a careful manner to avoid loss of optimality by stopping one of the levels of query too early—e.g., it may not be enough to stop a local search once any of the exits of the subgraph has been reached in the search.

We also believe that there is ample opportunity to developing further graph simplification techniques to prune the search for not entering in subgraphs that will never be part of the requested TDSP. For instance, a subgraph that is connected to the rest of the graph by a single node may be removed if the target node of the requested TDSP is not inside it. This would require a fast mechanism to identify such an exceptional case and simplify the graph "on-demand" before applying the TDSP problem solution. Though this should add some additional initialization computational effort, we certainly think that in some cases this might be worthwhile as would avoid the subsequent search in taking ways that do not lead anywhere else.

Perspectives on enhancing performance of the developed solutions

During the implementation of the case study some measures for improved performance were presented, some of them were even tested to prove the concept. Those were the trivial graph simplification by collapsing edges, but also a further simplification to allow fast traversal of a dense subgraph. Here we present further discussion on enhancing performance of the developed solutions taking mostly a perspective of existing speed-up techniques that were found in the literature (see Section 3.3).

The power of C-based programming languages. Programming languages that are of the group of C languages are well-known for their capability of handling heavy processing with high performance. Myers [61], for instance, acknowledges one of these languages (C++) as the one to use for "performance-critical" parts of solving a problem (or developing a system). In this sense, we believe some performance improvement could be achieved by simply transforming the culprit part of our developed algorithm, namely the update of arrival-time functions, into compiled C-based language code to be run as a library inside PostgreSQL. That perhaps would maintain the growth of computational complexity of this part of the algorithm under control and eventually turn it out to be possible of solving larger problems sizes and even in larger instances. We have tested this problem only with the higher hierarchy of roads. A C-based implementation can also help in the precomputation of least-delay functions for the graph simplification we have proposed, since this task suffers from the same problem of increasing data complexity as the update of arrival-time functions.

A*-based search. Another possibility is to make use of a technique to direct the search towards the target node such as the classic A* algorithm or any other variant of this principle (e.g., [47]). While in intuition this looks right, we have some concern whether this is always correct, particularly if the heuristics used is distance-based. Our approach over time-dependent routing is that perhaps any alternative route can be faster in traversing besides looking the shortest one distance-wise. Or in simpler words, how can one assure that there is no faster route to the target outside of the A*-based search space? Especially considering that giving up of optimality or computation of *exact* time-dependent shortest paths is not at all an option from our point of view. Hence, unless a heuristics can be developed on the basis of travel-time rather than distance and be provable to *always* provide optimal routes, our concern remains.

Hierarchical approaches. Somewhat the same concerns described for speed-up with use of A* search also apply here. With this we mean that a definition of hierarchy level purely based on distance (i.e., how far you can reach by using this road segment) may not suffice for a time-dependent setting. As a matter of fact, in the Netherlands, for instance, it is common sense to talk about the early morning peak time in the motorways and major roads just before people start their work duties. In this case, it seems obvious that motorways and major roads have great probability of being part of higher levels of any hierarchy-based partitioning of the graph as they tend to be important for long-distance travel. For instance, if we think of a hierarchy-based technique as "reach" [39] (see Section 3.3.2), logically those kinds of road often are on a shortest long-distance path. But those are also the roads that display biggest delays at peak traffic time. The question is then how is that fact taken into account when hierarchy levels? At least around peak time, it might well be that a driver is better off cutting through cities and auxiliary roads to avoid the traffic jam. We believe that a good approach for defining hierarchy level in a time-dependent setting cannot purely rely on distances, and occasionally this should be balanced with a measure of how frequent a road is at least close to a freeflow speed travel. In other words, that would be a hierarchy level definition which gives priority to roads that can be used for long-distance traffic, just as in the classical hierarchy-based techniques, but also with a refining parameter that indicates whether those roads are sensitive to jams or not.

Local + Global queries. In [10] a combination of local and global queries is devised for solving static shortest path with incredible performance results. As mentioned before, that could be a good approach to take advantages of the speed-up by substituting dense subgraphs for simplified versions (Section 5.6) and still be able to find a shortest path that start and/or finishes inside a simplified dense subgraph.

Other graph simplification techniques. To explore further techniques in graph simplification, such as by identifying exceptional subgraphs that do not need to be in a graph when searching a specific TDSP. This has to do with connectivity of graphs and we believe there is ample opportunity for enhancing performance by these means.

6.1.1 Further discussion—Looking outside the box

We here provide some further discussion on the issue of solving TDSP problems, particularly with focus on a database context. We remark that this setting is rarely found in the literature, where most reported approaches at best just mention a database as means to fetch the data, and run fully in main memory.

- It is rather difficult to compare results when the setting is completely different from others found in the literature. A main-memory solution diverges in behavior of one that runs in a database context. In our case, the solution was even implemented in a server scenario basis, meaning that occasionally performance may suffer from variables that are not under our control, such as the server's overload and perhaps limitations of hardware, given all the concurrent use. We do not think, however, that was an important impact in this project, but it may well be the case in such a setting.

- We acknowledge the non-triviality of solving TDSP in a database context using an approach that applies none heuristics to speed up computation for instances that appear in real-world applications. We believe that there is no other way of efficiently solving TDSP problems if not making use of speed-up techniques that come at the cost of precomputation and need for additional storage. In this sense, though there exists a trade-off to consider between these two issues and computational time to actually solve the problem, one could generally say that any precomputation that speeds up the solution considerably may well be accommodated when storage space is not a concern. In this direction, the work that has been carried out at Karlsruhe University deserves mention as they have a history of speed-up techniques for solving (static and dynamic) shortest path problems. Their achievements in static shortest paths has turned this kind of problem to be solvable in no more than several microseconds in continent-sized graph [10], as well as to claim to have the first technique (Time-dependent SHARC-Routing) to solve TDSP problems efficiently (in less than 160 seconds, but actually depending on how much traffic conditions variation is considered) over the same continent-sized graph [23].

- Some care has to be taken when analyzing reported computational results of speed-up techniques for solving shortest paths, since in many cases the query times achieved only consider the solution of the problem without actually outputting the path through the original graph itself (e.g., [80, 23]).

6.2 Conclusions

With respect to the objectives defined for this research, we draw the following conclusions:

- The translation of the time-dependent network for the Netherlands was conducted by abstracting a graph without its geometric layout, i.e., only presenting the topological relationships between nodes and edges, but not having their spatial dimension. That was preferred to avoid overloading the solutions with data that is not needed in a graph-theoretic approach. As the time-dependency of the edges is presented as speed profiles, we had essentially two options for delivering the real costs (i.e., travel times) to the edges in the graph: (1) to compute edge-delay functions "on-demand" or (2) to precompute edge-delay functions. The second option was obviously identified as preferable because of performance.

- Efficiently solving TDSP problem in a database context has shown to be not a trivial task, and a completely functional approach may well be beyond the scope and time of an MSc project. Particularly, if we consider the choice we made from the beginning of using an approach not based on any heuristics or speed-up technique for solving the problem. It turned out that the developed solutions are far from providing high-performing query responses for TDSP, especially if applied over the original graph and a more complex TDSP-LTT request is made. In any case, we might say that the research project helped in achieving a sort of benchmark for solving TDSP in a database context, against which further works in speeding up the solutions can be compared.

- Two main issues were identified as performance bottlenecks for solving TDSP-LTT: one general concern for TDSP computations and another approach-specific. The first one is the growth in complexity of the arrival-time functions (i.e., the potential increase in numbers of concrete points in arrival-time functions), which eventually is responsible for a loss in performance per iteration as the problem worsens. The second one is the slowdown in convergence of the departure time interval for optimization when the search reaches parts of the graph characterized by small delay-values in the edge-delay functions.

- The chosen approach, namely Two-step LTT, was shown to outperform some of its counterparts in experimental tests conducted in the study in which it was proposed, when edge-delay functions were arbitrarily generated [28]. Against real-world edge-delay functions in our case, however, it has shown to have a feature that is not usually desirable for any algorithmic approach: to be fairly dependent on the characteristics of the input data. Here, we refer to the slowdown in the convergence of the departure time interval as mentioned in Section 5.5.1.

- Facing the somewhat unimpressive performance of the developed solutions, we conceptually defined possible ways of coping with some of the shortcomes. We first introduced the trivial simplification of the graph that removes unnecessary nodes of degree two and collapses the path through these nodes into a single edge. That is also referred to as defining the 2-core of graph in the literature [23]. This simplification positively affected the performance of the simple TDSP-GDT, but was not enough to make the more complex TDSP-LTT solvable except for rather small request sizes. We revisited then the simplification of the graph by conceptually defining a possible way of speeding up computations in the context of TDSP-LTT problems, and we introduced algorithmic approaches to devise the envisaged graph simplification. That was conceived in a manner that attempts to comply with the general rule that a precomputation task should not require more than linear growth in storage for the precomputed data when compared to the input graph.

- A proof-of-concept performance test showed that the introduced graph simplification can considerably speed up the computation of TDSP using the chosen approach. That is because by this means short edges (with small delay-values) are removed from the graph, thus the algorithm does not slow down in the convergence of the interval for departure time. We envisage two possibilities for using this speed-up technique: (1) by quickly substituting precomputed simplified versions of dense subgraphs; the solution would be applied over a single graph with parts substituted; (2) by having two levels of search, *local* and *global*, in which the global query would be applied over the simplified graph when far enough from start/target node, while the local query would take over when the search is close enough to start/target node. Obviously, the local query would be applied over the original graph with only the trivial simplification introduced in Section 5.4.2.

- Unfortunately, due to the faced problems with performance of the computations, it was not possible to conduct a thorough set of performance tests, apart from the run tests to verify the solutions developed. That would allow to check the scalability of the approaches for different-sized networks. The reason behind this is the realization that it does not make sense to verify scalability when the developed solutions in general show bad performance even for a rather sparse network as used in the run tests. We then invested more time in deriving speeding-up techniques as described above.

6.3 Recommendations

Taking into account the results achieved in this research project, the following recommendations are made for further improvement:

- As it is clear that solving TDSP problems without making use of any heuristics to speed up computations is rather unfeasible, we propose further research on techniques to realize faster computations of routes in a database context. This could start with materializing the concepts and algorithmic approaches proposed in this research project (i.e., simplification of the graph in dense subgraphs inside of it) and verifying its applicability in practice as well as its potential of speed-up in computations of TDSP problems.

- To study, understand and implement approaches proposed in the literature, with a special note for SHARC-Routing algorithm in [23] that has been already adapted for time-dependent networks. In this matter, it seems fruitful to contact the research group at the Institute for Theoretical Computer Science, Karlsruhe University (Germany), which has put a lot of effort in delivering fast solutions for real-world applications with routing in transportation networks.

- To consider the issues discussed under the heading “Perspectives on enhancing performance of the developed solutions” of this chapter, in which we have put into context possible avenues to exploit and eventually deliver faster computations of time-dependent routes.

- We particularly think that solutions of TDSP problems in considerably large graphs can only be achieved by working under a *hierarchical* perspective in which the search for shortest paths would be restricted to a rather sparse network of high-hierarchy roads when start and target node are fairly apart. A careful study of the concepts of *local* and *global queries* can be worthwhile to separate searches that need to look at lower levels of hierarchy from those that can be sped-up in the sparse network of high-hierarchy roads. An interesting discussion in this sense is a proper definition of hierarchies in time-dependent routing as discussed above (see “Perspectives on enhancing performance of the developed solutions” in this chapter).

- When storage of precomputed data is not a limiting factor for the solution being developed, an extreme speed-up measure for long-distance travels is to define a (fairly small) set of proper *transit nodes* to which all-pairs of *least-delay functions* would are beforehand, similarly to what happens in the fastest static shortest path computation method known [10]. In this case, the real computation expense is restricted to local queries (if applicable), in which only a considerably small subset of the graph can be necessary for the search.

Bibliography

[1] R. K. Ahuja, J. B. Orlin, S. Pallottino, and M. G. Scutella. Dynamic shortest paths minimizing travel times and costs. *Working Paper Series SSRN*, 2002.

[2] S. Ambler. *Agile Database Techniques: Effective Strategies for the Agile Software Developer*. Wiley Publishing, 2003.

[3] S. Arora. Polynomial time approximation schemes for euclidean traveling salesman and other geometric problems. *J. ACM*, 45(5):753–782, 1998.

[4] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The object-oriented database system manifesto, 1989.

[5] T. Atlas. Tele Atlas Multinet shapefile 4.4. format specifications. Technical report, Tele Atlas and Tele Atlas North America, 2008.

[6] T. Atlas. Tele Atlas Speed Profiles: intelligent data for optimal routing. specifications 2.0. Technical report, Tele Atlas and Tele Atlas North America, 2008.

[7] T. Atlas. Tele Atlas Speed Profiles: intelligent data for optimal routing. user guide 2008.10. Technical report, Tele Atlas and Tele Atlas North America, 2008.

[8] S. Balev, F. Guinand, and Y. Pigné. Maintaining shortest paths in dynamic graphs. In *International Conference on Non-Convex Programming: local and global approaches. Theory, Algorithms and Applications*, 2007.

[9] H. Bast, S. Funke, and D. Matijevic. Transit: Ultrafast shortest-path queries with linear-time preprocessing. In *9th DIMACS Implementation Challenge — Shortest Path*, pages 66–79. DIMACS, 2006.

[10] H. Bast, S. Funke, P. Sanders, and D. Schultes. Fast routing in road networks using transit nodes. *Science*, 316(5824):566, April 2007.

[11] R. Bauer and D. Delling. SHARC: Fast and Robust Unidirectional Routing. In *Proceedings of the 10th Workshop onAlgorithm Engineering and Experiments (ALENEX'08)*. SIAM, 2008. to appear.

[12] R. Bauer and D. Delling. Sharc: Fast and robust unidirectional routing. *J. Exp. Algorithmics*, 14:2.4–2.29, 2009.

[13] D. Bertsimas and D. Simchi-Levi. A new generation of vehicle routing research: Robust algorithms, addressing uncertainty. *Operations Research*, 44(2):286–304, 1996.

[14] J.-F. Bérubé, J.-Y. Potvin, and J. G. Vaucher. Time-dependent shortest paths through a fixed sequence of nodes: application to a travel planning problem. *Computers & OR*, 33:1838–1856, 2006.

[15] J. A. Bondy and U. S. R. Murty. *Graph Theory with Applications*. The Macmillan Press, first edition reprinted edition, 1978.

[16] I. Chabini. Discrete dynamic shortest path problems in transportation applications: Complexity and algorithms with optimal run time. *Transportation Research Records*, 1645:170–175, 1998.

[17] S. Cook. The P versus NP problem. manuscript prepared for the clay mathematics institute for the millennium prize problems, 2000.

[18] J. F. Cordeau, M. Gendreau, G. Laporte, J. Y. Potvin, and F. Semet. A guide to vehicle routing heuristics. *The Journal of the Operational Research Society*, 53(5):512–522, 2002.

[19] B. C. Dean. Continuous-time dynamic shortest path algorithms. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, May 1999.

[20] B. C. Dean. Shortest paths in FIFO time-dependent networks: theory and algorithms. Technical report, MIT, Cambridge, MA, 2004.

[21] M. Dell'Amico, M. Iori, and D. Pretolani. Shortest paths in piecewise continuous time-dependent networks. *Operations Research Letters*, 36(6):688 – 691, 2008.

[22] D. Delling. Time-dependent sharc-routing. In *ESA'08: Proceedings of the 16th annual European symposium on Algorithms*, pages 332–343, Berlin, Heidelberg, 2008. Springer-Verlag.

[23] D. Delling. *Engineering and Augmenting Route Planning Algorithms*. PhD thesis, Universität Fridericiana zu Karlsruhe (TH), Hamburg, Germany, February 2009.

[24] D. Delling and G. Nannicini. Core routing on dynamic time-dependent road networks. In *Journal Version of ISAAC'08*, 2008.

[25] D. Delling, P. Sanders, D. Schultes, and D. Wagner. Engineering route planning algorithms. In *Algorithmics of Large and Complex Networks: Design, Analysis, and Simulation*, pages 117–139, Berlin, Heidelberg, 2009. Springer-Verlag.

[26] D. Delling and D. Wagner. Landmark-Based Routing in Dynamic Graphs. In *Proceedings of the 6th Workshop on Experimental Algorithms (WEA'07)*, volume 4525 of *Lecture Notes in Computer Science*, page 5265. Springer, June 2007.

[27] B. Ding, J. X. Yu, and L. Qin. Finding time-dependent shortest paths over large graphs. In *EDBT '08: Proceedings of the 11th international conference on Extending database technology*, pages 205–216, New York, NY, USA, 2008. ACM.

[28] L. Everett, L.-S. Wang, and S. Hannenhalli. Dense subgraph computation via stochastic search: application to detect transcriptional modules. In *ISMB (Supplement of Bioinformatics)*, pages 117–123, 2006.

[29] P. Festa. Shortest path algorithms. In *Handbook of Optimization in Telecommunications*, pages 185–210, Berlin, Heidelberg, 2006. Springer US.

[30] B. Fleischmann, M. Gietz, and S. Gnutzmann. Time-varying travel times in vehicle routing. *Transportation Science*, 38(2):160–173, 2004.

[31] B. Fleischmann, S. Gnutzmann, and E. Sandvo"s. Dynamic vehicle routing based on online traffic information. *Transportation Science*, 38(4):420–433, 2004.

[32] L. Fortnow and S. Homer. A short history of computational complexity. In *The History of Mathematical Logic*. North-Holland, 2002.

[33] S. Gao and I. Chabini. Optimal routing policy problems in stochastic time-dependent networks. *Transportation Research Part B: Methodological*, 40(2):93 – 122, 2006.

[34] D. Gibson, R. Kumar, and A. Tomkins. Discovering large dense subgraphs in massive graphs. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 721–732. VLDB Endowment, 2005.

[35] A. V. Goldberg and C. Harrelson. Computing the shortest path: A* search meets graph theory. In *SODA'05: In Proceedings of the 16th Annual ACM–SIAM Symposium on Discrete Algorithms*, pages 156–165, 2005.

[36] A. V. Goldberg, H. Kaplan, and R. F. Werneck3. Reach for a*: Efficient point-to-point shortest path algorithms. Technical report, Microsoft Research, October 2006.

[37] R. J. Gutman. Reach-based routing: A new approach to shortest path algorithms optimized for road networks. In *ALENEX/ANALC*, pages 100–111, 2004.

[38] H. W. Hamacher, S. Ruzika, and S. A. Tjandra. Algorithms for time-dependent bicriteria shortest path problems. *Discrete Optimization*, 3(3):238–254, 2006.

[39] S. Hanson. Off the road? Reflections on transportation geography in the information age. *Journal of Transport Geography*, 6(4):241–249, 1998.

[40] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(8), 1968.

[41] U. Helling and J. Schoenharting. Travel time advantages by dynamic route guidance in germany: Status quo and improvement potential. In *European Transport Conference 2006*. AET, 2006.

[42] M. E. Horn. On-line vehicle routing and scheduling with time-varying travel speeds. *Journal of Intelligent Transportation Systems*, 10(1):160–173, 2006.

[43] C.-I. Hsu, S.-F. Hung, and H.-C. Li. Vehicle routing problem with time-windows for perishable food delivery. *Journal of Food Engineering*, 80(2):465 – 475, 2007.

[44] B. Huang and X. Pan. Gis coupled with traffic simulation and optimization for incident response. *Computers, Environment and Urban Systems*, 31(2):116 – 132, 2007.

[45] B. Huang, Q. Wu, and F. B. Zhan. A shortest path algorithm with novel heuristics for dynamic transportation networks. *International Journal of Geographical Information Science*, 21(6):625 – 644, 2007.

[46] R. Impagliazzo. A personal view of average-case complexity. In *SCT '95: Proceedings of the 10th Annual Structure in Complexity Theory Conference (SCT'95)*, page 134, Washington, DC, USA, 1995. IEEE Computer Society.

[47] P. Ji, Y. Z. Wu, H. Z. Liu, and H. T. Wu. The vehicle routing problem with time-varying travel times and a solution method. *International Journal of Innovative Computing Information and Control*, 5(4):1001–1011, Apr 2009.

[48] E. Kanoulas, Y. Du, T. Xia, and D. Zhang. Finding fastest paths on a road network with speed patterns. In *ICDE '06: Proceedings of the 22nd International Conference on Data Engineering*, page 10, Washington, DC, USA, 2006. IEEE Computer Society.

[49] D. E. Kaufman and R. L. Smith. Fastest paths in time-dependent networks for intelligent vehicle-highway systems application. In *IVHS Journal*, pages 1–11. Taylor & Francis, 1993.

[50] S. Koenig, M. Likhachev, and D. Furcy. Lifelong Planning A*. *Artif. Intell.*, 155(1-2):93–146, 2004.

[51] A. Larsen. *The Dynamic Vehicle Routing Problem*. PhD thesis, Technical University of Denmark, June 2000.

[52] A. Larsen, O. Madsen, and M. Solomon. Partially dynamic vehicle routing-models and algorithms. *The Journal of the Operational Research Society*, 53(6):637–646, 2002.

[53] L. Liu and L. Meng. Algorithmic concerns of multi-modal route planning. In *LBS Telecartography: eletronic proceedings of the 5th symposium on Location Based Services & Telecartography*, pages 19–25, Salzburg, Austria, 2008.

[54] W. Maden, R. W. Eglese, and D. Black. Vehicle routing and scheduling with time varying data: A case study. *Lancaster University Management School Working Papers*, 2009.

[55] O. B. Madsen, A. Larsen, and M. M. Solomon. Recent developments in dynamic vehicle routing systems. In B. L. Golden, S. Raghavan, and E. A. Wasil, editors, *The Vehicle Routing Problem*, pages 199–218. Springer Science+Business Media, New York, NY, 2008.

[56] C. Malandraki and M. S. Daskin. Time Dependent Vehicle Routing Problems: Formulations, Properties and Heuristic Algorithms. *Transportation Science*, 26(3):185–200, 1992.

[57] J. Maue, P. Sanders, and D. Matijevic. Goal-directed shortest-path queries using precomputed cluster distances. *J. Exp. Algorithmics*, 14:3.2–3.27, 2009.

[58] N. C. Myers. C++ in the real world: Advice from the trenches, 1997. [Online; accessed 20-January-2010].

[59] G. Nannicini, P. Baptiste, D. Krob, and L. Liberti. Fast point-to-point shortest path queries on dynamic road networks with interval data. In *CTW*, pages 115–118, 2007.

[60] G. Nannicini, D. Delling, L. Liberti, and D. Schultes. Bidirectional A* on time-dependent graphs. In *CTW*, pages 132–135, 2008.

[61] G. Nannicini and L. Liberti. Shortest paths in dynamic graphs. *International Transactions in Operational Research*, 15:551–563, 2008.

[62] Y. M. Nie and X. Wu. Shortest path problem considering on-time arrival probability. *Transportation Research Part B: Methodological*, 43(6):597 – 613, 2009.

[63] A. Orda and R. Rom. Shortest-path and minimum-delay algorithms in networks with time-dependent edge-length. *Journal of the Association for Computing Machinery*, 37(3):607–625, 1990.

[64] A. Osvald and L. Z. Stirn. A vehicle routing algorithm for the distribution of fresh vegetables and similar perishable food. *Journal of Food Engineering*, 85(2):285 – 295, 2008.

[65] D. Pfoser, S. Brakatsoulas, P. Brosch, M. Umlauft, N. Tryfona, and G. Tsironis. Dynamic travel time provision for road networks. In *GIS '08: Proceedings of the 16th ACM SIGSPATIAL international conference on Advances in geographic information systems*, pages 1–4, New York, NY, USA, 2008. ACM.

[66] D. Pfoser, N. Tryfona, and A. Voisard. Dynamic travel time maps: Enabling efficient navigation. In *Proc. 18th SSDBM conf.*, pages 369–378, 2006.

[67] J.-Y. Potvin, Y. Xu, and I. Benyahia. Vehicle routing and scheduling with dynamic travel times. *Comput. Oper. Res.*, 33(4):1129–1137, 2006.

[68] H. N. Psaraftis. Dynamic vehicle routing problems. In B. L. Golden and A. A. Assad, editors, *Vehicle Routing: Methods and Studies*, chapter 11, pages 223–248. Elsevier Science Publishers B.V., Amsterdam, North-Holland, 1988.

[69] H. N. Psaraftis. Dynamic vehicle routing: Status and prospects. *Journal of Heuristics*, 61(1):143–164, 1995.

[70] H. N. Psaraftis and J. N. Tsitsiklis. Dynamic shortest paths in acyclic networks with markovian arc costs. *Oper. Res.*, 41(1):91–101, 1993.

[71] N. Rigo, R. Hekkenberg, A. B. Ndiaye, D. Hadhazi, G. Simongati, and C. Hargitai. Performance assessment for intermodal chains. *European Journal of Transport and Infrastructure Research*, 7(4):283–300, 2007.

[72] H. E. Romeijn and R. L. Smith. Parallel algorithms for solving aggregated shortest-path problems. *Comput. Oper. Res.*, 26(10-11):941–953, 1999.

[73] P. Sanders and D. Schultes. Highway hierarchies hasten exact shortest path queries. In *ESA*, pages 568–579, 2005.

[74] P. Sanders and D. Schultes. Engineering highway hierarchies. In *ESA'06: Proceedings of the 14th conference on Annual European Symposium*, pages 804–816, London, UK, 2006. Springer-Verlag.

[75] D. Schrank and T. Lomax. 2009 urban mobility report. Technical report, Texas Transportation Institute, Texas A&M University System, College Station, Texas, July 2009.

[76] D. Schultes. *Route Planning in Road Networks*. PhD thesis, Universität Karlsruhe (TH), February 2008.

[77] D. Schultes and P. Sanders. Dynamic highway-node routing. In *WEA'07: Proceedings of the 6th Workshop on Experimental Algorithms*, pages 66–79. Springer, 2007.

[78] R. Séguin, J. Y. Potvin, M. Gendreau, T. G. Crainic, and P. Marcotte. Real-time decision problems: An operational research perspective. *The Journal of the Operational Research Society*, 48(2):162–174, 1997.

[79] R. Sommar and J. Woxenius. Time perspectives on intermodal transport of consolidated cargo. *European Journal of Transport and Infrastructure Research*, 7(2):163–182, 2007.

[80] K. Sung, M. G. H. Bell, M. Seong, and S. Park. Shortest paths in a network with time-dependent flow speeds. *European Journal of Operational Research*, 121(1):32 – 39, 2000.

[81] T. van Woensel, L. Kerbache, H. Peremans, and N. Vandaele. Vehicle routing with dynamic travel times: A queueing approach. *European Journal of Operational Research*, 186(3):990 – 1007, 2008.

[82] D. Wagner, T. Willhalm, and C. Zaroliagis. Geometric containers for efficient shortest-path computation. *J. Exp. Algorithmics*, 10:1.3, 2005.

[83] Q. Wu, J. Hartley, and D. Al-Dabass. Time-dependent stochastic shortest path(s) algorithms for a scheduled transportation network. *I.J. of Simulation*, 6(7-8), 2005.

[84] X. Yin, Z. Ding, and J. Li. A shortest path algorithm for moving objects in spatial network databases. *Progress in Natural Science*, 18(7):893 – 899, 2008.

[85] M. L. Yiu and N. Mamoulis. Clustering objects on a spatial network. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 443–454, New York, NY, USA, 2004. ACM.

[86] J. Yoon, B. Noble, and M. Liu. Surface street traffic estimation. In *MobiSys '07: Proceedings of the 5th international conference on Mobile systems, applications and services*, pages 220–232, New York, NY, USA, 2007. ACM.

[87] F. B. Zhan and C. E. Noon. Shortest path algorithms: an evaluation using real road networks. *Transportation Science*, 32(1):65–73, 1998.

[88] L. Zhao and T. Ohshima. A* algorithm for the time-dependent shortest path problem. In *WAAC08: The 11th Japan-Korea Joint Workshop on Algorithms and Computation*, 2008.

Appendix A

Preparation of dataset for case study

A.1 Loading GIS dataset into the database

```
---creating the sql insert command file (+ create table)

shp2pgsql -I -S -g geom -s 4326 filename.shp schema.tablename >
sqlfilename.sql

--filename.shp -> nldnld________nw.shp (road segments);
--                nldnld________jc (road junctions)

---load into the database
psql -h itcnt07.itc.nl -d dynamic_routing -f sqlfilename.sql
-U rodrigo
```

A.2 Loading time-dependency of the road segments

```
---speed profiles table
create table speed_profile
(profile_id integer not null,
time_slot integer not null,
rel_sp real not null,
constraint speed_profile_pk primary key (profile_id,time_slot)
);

---inserting speed profiles data into table
psql => \copy "speed_profile" from filename.txt

---speed profile function table
create table speed_profile_function
(pid integer not null,
constraint speed_profile_function_pk primary key (pid)
);

select addgeometrycolumn
('dynamic_routing','speed_profile_function','geom','-1',
'linestring',2);

insert into speed_profile_function
select sp.profile_id, st_linefrommultipoint(
st_collect(
st_snaptogrid(
st_makepoint(sp.time_slot,sp.rel_sp),0.1,0.1))))
from speed_profile as sp
group by sp.profile_id;

---hsnp link table
create table nldnld_hsnp
(
  network_id numeric(15) not null,
  val_dir integer not null,
  spfreeflow integer,
  spweekday integer,
  spweekend integer,
  profile_1 integer,
  profile_2 integer,
  profile_3 integer,
  profile_4 integer,
  profile_5 integer,
  profile_6 integer,
  profile_7 integer,
  constraint hsnp_pk primary key (network_id, val_dir)
);

---inserting hsnp link table data into table
psql => \copy "nldnld_hsnp" from filename.txt
```

A.3 Abstracting the graph

```
---creating vertical subset of nldnld_nw
create table nw_edges
(
  id bigint,
  f_jnctid bigint,
   t_jnctid bigint,
  meters double precision,
  oneway character varying(2),
  kph smallint,
   constraint nw_edges_pkey primary key (id)
);

insert into nw_edges
select n.id,n.f_jnctid,t_jnctid,n.meters,n.oneway,n.kph
from nldnld_nw as n
where n.oneway<>'N' or n.oneway is null;

---extracting edges of motorways and major roads
create table edge_majroad
(id bigint not null,
 val_dir smallint not null,
 f_node bigint,
 t_node bigint,
 length double precision,
 constraint edge_pk1 primary key (id,val_dir));

insert into edge_majroad
--geometry direction corresponds to graph direction
select e.id, 2 as val_dir, e.f_jnctid as f_node, e.t_jnctid as
t_node, e.meters as length
from nw_edges e
where e.oneway::text = 'FT'::text and (e.frc=0 or e.frc=1)
union all
--geometry direction does not correspond to graph direction

select e.id, 3 as val_dir, e.t_jnctid as f_node, e.f_jnctid as
```

```
t_node, e.meters as length
from nw_edges e
where e.oneway::text = 'TF'::text and (e.frc=0 or e.frc=1)
union all
--biderectional geometry
select e.id, 3 as val_dir, e.t_jnctid as f_node, e.f_jnctid as
t_node, e.meters as length
from nw_edges e
where e.oneway is null and (e.frc=0 or e.frc=1)
union all
select e.id, 2 as val_dir, e.f_jnctid as f_node, e.t_jnctid as
t_node, e.meters as length
from nw_edges e
where e.oneway is null and (e.frc=0 or e.frc=1);

create unique index edge_majroad_idx on edge_majroad (id,val_dir);

---extracting nodes of motorways and major roads
create table node_majroad
(id bigint,
 constraint node_pkey primary key (id)
);

insert into node_majroad
select distinct f_node as id
from edge_majroad
union
select distinct t_node
from edge_majroad;
```

Appendix B

Solving TDSP problems in a database context

B.1 Auxiliary tools

B.1.1 Initialization tool—TDSP-GDT

```
----creating arrival-time functions + edge-delay functions

create or replace function dr_initialize(node_table varchar,
edge_table varchar, vs bigint, ve bigint, ts double precision, dw integer)
  returns void as
$body$
---node_table is the table that contains the set of nodes of the graph;
---edge_table is the table that contains the set of edges of the graph;
---vs is the source node id of the path to be found;
---ve is the target node id of the path to be found;
---dw is the day of the week for the traverse;
declare
edg record;
countq integer;
ltt arrival_time_function;
begin
create table _queue_arr_time_
(vid bigint not null,
tau double precision not null,
at_tau double precision not null,
--active boolean,
constraint _queue_arr_time__pk1 primary key (vid,tau));
create table _arrivaltime_
(vid bigint not null,
"time" double precision not null,
arrivaltime double precision not null,
--active boolean,
constraint _arrivaltime__pk1 primary key (vid,"time"));
create table _edgedelay_function_
(eid bigint not null,
val_dir integer not null,
constraint _edgedelay_function_pk primary key (eid,val_dir)
);
perform addgeometrycolumn
('dynamic_routing','_edgedelay_function_','geom','-1','linestring',2);
create index edgedelay__geom_gist
  on _edgedelay_function_
  using gist
  (geom);
```

```
insert into _arrivaltime_ values (vs,ts,ts);
execute
'insert into _arrivaltime_
select n.id, '|| ts::text || ', (' || ts::text || '+1)*90000
from ' || quote_ident(node_table) || ' as n
where n.id <> ' || vs::text;
---creating edge-delay functions for proper day of the week
for edg in execute 'select * from' || quote_ident(edge_table) ||
loop
insert into _edgedelay_function_
select edg.id, edg.val_dir, st_snaptogrid(st_scale(
                                          st_simplify(s.geominv,0),
                                          1,edg.length*3.6/n.spfreeflow
                                          ),0,0.01)
from (select case when p.spfreeflow > 0 then p.spfreeflow else
case when dw = 1 or dw = 7 then p.spweekend else p.spweekday end
           end as spfreeflow,
(case
when dw = 1 then case when p.profile_1 = 0 then 1 else p.profile_1 end
when dw = 2 then case when p.profile_2 = 0 then 1 else p.profile_2 end
when dw = 3 then case when p.profile_3 = 0 then 1 else p.profile_3 end
when dw = 4 then case when p.profile_4 = 0 then 1 else p.profile_4 end
when dw = 5 then case when p.profile_5 = 0 then 1 else p.profile_5 end
when dw = 6 then case when p.profile_6 = 0 then 1 else p.profile_6 end
when dw = 7 then case when p.profile_7 = 0 then 1 else p.profile_7 end
end) as profile
       from nldnld_hsnp as p
       where edg.id=p.network_id and edg.val_dir=p.val_dir) as n,
       speed_profile_function as s
where s.pid=n.profile;
end loop;
insert into _queue_arr_time_
select * from _arrivaltime_ as at;
end;
$body$
  language 'plpgsql' volatile strict
  cost 1;

----creating only arrival-time functions

create or replace function dr_initialize(node_table character varying,
edgedelay_table varchar, vs bigint, ve bigint, ts double precision,
dw integer)
  returns void as
$body$
---node_table is the table that contains the set of nodes of the graph;
---edgedelay_table is the table with precomputed edge-delay functions;
---vs is the source node id of the path to be found;
---ve is the target node id of the path to be found;
---dw is the day of the week for the traverse;
declare
edg record;
countq integer;
ltt arrival_time_function;
begin
create table _queue_arr_time_
(vid bigint not null,
tau double precision not null,
at_tau double precision not null,
constraint _queue_arr_time__pk1 primary key (vid,tau));
create table _arrivaltime_
(vid bigint not null,
"time" double precision not null,
arrivaltime double precision not null,
constraint _arrivaltime__pk1 primary key (vid,"time"));
create table _edgedelay_function_
(eid bigint not null,
val_dir integer not null,
constraint _edgedelay_function_pk primary key (eid,val_dir)
);
perform addgeometrycolumn
('dynamic_routing','_edgedelay_function_','geom','-1','linestring',2);
create index edgedelay__geom_gist
  on _edgedelay_function_
  using gist
  (geom);
insert into _arrivaltime_ values (vs,ts,ts);
execute
'insert into _arrivaltime_
select n.id, '|| ts::text || ', (' || ts::text || '+1)*90000
from ' || quote_ident(node_table) || ' as n
where n.id <> ' || vs::text;
---fecthing precomputed edge-delay functions for proper day of the week
execute
'insert into _edgedelay_function_
select eid, val_dir, geom_'  || dw::text || '
from ' || quote_ident(edgedelay_table);
insert into _queue_arr_time_
select * from _arrivaltime_ as at;
end;
$body$
  language 'plpgsql' volatile strict
  cost 1;
```

B.1.2 Initialization tool—TDSP-LTT

```
create or replace function dr_initialize(
node_table character varying,
edgedelay_table character varying, vs bigint,
ve bigint, ts double precision,
te double precision, dw integer)
  returns void as
$body$
---edgedelay_table is the table with the edge-delay functions;
---node_table is the table with the nodes of the graph;
---vs start node id;
---ve target node id;
---ts initial time of interval for departure time;
---te final time of interval for departure time;
---dw day of week;
declare edg record; ltt arrival_time_function; begin insert into
arrival_time_function values (vs,dr_source_g_function(ts,te));
execute 'insert into arrival_time_function select
n.vid,dr_others_g_function(' || ts::text || ',' || te::text || ')
from ' || quote_ident(node_table) ||' as n where n.vid <>' ||
vs::text; raise notice 'i finished to add arrival_time_function!';
create table _queue_arr_time_ (vid bigint not null, tau double
precision not null, at_tau double precision not null, constraint
_queue_arr_time__pk1 primary key (vid,tau,at_tau)); create table
_control_node_subinterval_ (vid bigint not null, time_s double
precision not null, time_e double precision not null, constraint
_control_node_subinterval__pk1 primary key (vid)); create table
_subset_edgedelay_function_ (eid bigint not null, val_dir integer
not null, constraint _subset_edgedelay_function_pk primary key
(eid,val_dir) ); perform addgeometrycolumn
('dynamic_routing','_subset_edgedelay_function_',
'tempgeom','-1','linestring',2); create index edge_delay__geom_gist
  on _subset_edgedelay_function_
  using gist
  (tempgeom);
for ltt in select * from arrival_time_function
```

```
loop
  insert into _queue_arr_time_
  select ltt.vid, st_x(st_pointn(ltt.geom,n)), st_y(st_pointn(ltt.geom,n))
  from generate_series(1,st_numpoints(ltt.geom)) as h(n);
end loop;
execute 'insert into _control_node_subinterval_
select n.vid,'|| ts::text ||','|| te::text ||
' from ' || quote_ident(node_table) || ' as n
where n.vid = ' || vs::text;
execute 'insert into _control_node_subinterval_
select n.vid,' || 0::text || ',' || 0::text ||
' from ' || quote_ident(node_table) ||' as n
where n.vid <>' || vs::text;
execute
'insert into _subset_edgedelay_function_
select eid, val_dir, geom_'  || dw::text || '
from ' || quote_ident(edgedelay_table);
end;
$body$
  language 'plpgsql' volatile strict
  cost 1;

---creating initial arrival-time functions tools

--all nodes in the graph except for start node:
create or replace function dr_others_g_function(
double precision, double precision)
  returns geometry as
$body$
---$1 ts - initial interval time;
$2 te - end interval time;
$3 - node
select st_linefrommultipoint(st_collect(st_makepoint($1,($1+1)*90000),
st_makepoint($2,($2+1)*90000))) as geom
---90000 is something big enough to be considered 'infinity'
$body$
  language 'sql' immutable strict
  cost 1;

--start node:
create or replace function dr_source_g_function(
double precision, double precision)
  returns geometry as
$body$
---$1 ts - initial interval time;
$2 te - end interval time;
$3 - start node
select st_linefrommultipoint(st_collect(st_makepoint($1,$1),
st_makepoint($2,$2))) as geom
$body$
  language 'sql' immutable strict
  cost 1;
```

B.1.3 Look-up tools

```
---Look-up X-value
create or replace function dr_max_startingtime_value(
geometry, double precision,
double precision)
  returns double precision as
$body$
select case when
st_interSection($1,st_geometryfromtext(
'linestring(' || '0 ' ||
($2::text) || ','  || '1e37 ' || ($2::text) || ')')) =
'geometrycollection empty'
then $3
else
st_xmin(st_interSection($1,st_geometryfromtext(
'linestring(' || '0 ' ||  ($2::text) || ','  || '1e37 ' ||
($2::text) || ')')))
end
$body$
  language 'sql' immutable strict
  cost 1;

---Look-up Y-value
create or replace function dr_delay_value(
geometry, double precision)
  returns double precision as
$body$
select st_ymin(st_interSection(st_linefrommultipoint(
  st_geometryfromtext('multipoint(' || ($2::text) || ' 0,' ||
($2::text) || ' 1e+37)')),$1))
$body$
  language 'sql' immutable strict
  cost 1;
```

B.1.4 Update arrival-time functions tool

This couples the tools for add-up and to get the minimal of two functions.

```
create or replace function dr_arrivaltime_function(
geometry, geometry, geometry, double precision, double precision)
  returns geometry as
$body$
---$1 arrival-time function of the node in iteration;
---$2 edge-delay function of the edge to be traversed;
---$3 arrival-time function of the node to be updated;
---$4/$5 start/end of interval of time to update
select st_linefrommultipoint(st_collect(st_makepoint(x.t,x.atnew)))
from (select q.t, case when (q.t >= $4 and q.t <= $5) then
                       case when (dr_delay_value($1,q.t) +
                       dr_delay_value($2,dr_delay_value($1,q.t)))
                       < dr_delay_value($3,q.t) then
                       (dr_delay_value($1,q.t) +
                       dr_delay_value($2,dr_delay_value($1,q.t)))
                       else dr_delay_value($3,q.t) end
                  else  dr_delay_value($3,q.t) end as atnew
      from (select st_x(st_pointn($1,n)) as t
        from generate_series(1,st_numpoints($1)) as h(n)
        where  st_x(st_pointn($1,n)) >= $4 and st_x(st_pointn($1,n)) <= $5
        union
        select st_x(st_pointn($2,n)) as t
        from generate_series(1,st_numpoints($2)) as h(n)
            where  st_x(st_pointn($2,n)) >= $4 and
            st_x(st_pointn($2,n)) <= $5
            union
        select st_x(st_pointn($3,n)) as t
        from generate_series(1,st_numpoints($3)) as h(n)
        union
        select $5 as t
            ) as q
      order by q.t) as x
$body$
  language 'sql' immutable strict
  cost 1;
```

B.2 Two-step LTT for TDSP-GDT

B.2.1 Time-refinement step—TDSP-GDT

```
create or replace function dr_time_refinement_sg(edge_table varchar,
 vs bigint, ve bigint, ts double precision, dw integer)
  returns void as
$body$
---edge_table is the table with the edges of the graph;
---vs start node id;
---ve end node id;
---ts initial time of interval for departure time;
---te final time of interval for departure time;
---dw day of week;
declare
edg record; ltt record; tau_i double precision; g_ti double
precision; tau_k double precision; g_tk double precision; n_i
bigint; n_k bigint; countq integer; upd record; countupd integer;
begin tau_i := ts; tau_k := tau_i; countq = 0; while (select
count(*)
       from _queue_arr_time_ as q
       ) >= 2 loop
countq = countq + 1; select q.vid, q.at_tau, q.tau
  from _queue_arr_time_ as q
  order by q.at_tau
  limit 1 into n_i,g_ti,tau_i;
delete from _queue_arr_time_ where vid=n_i; select q.vid, q.at_tau,
q.tau
  from _queue_arr_time_ as q
  where q.vid <> n_i ---and q.active = true
  order by q.at_tau
  limit 1 into n_k,g_tk, tau_k;
for ltt in execute 'select * from _arrivaltime_ as at inner join '
|| quote_ident(edge_table) || ' as e on e.t_node=at.vid where
e.f_node=' || n_i::text loop
  select into upd n.vid, g_ti + dr_delay_value(n.geom,g_ti) as newat
          from (select ltt.t_node as vid, ef.tempgeom as geom
                from _edgedelay_function_ as ef
                where ltt.id=ef.eid and ltt.val_dir=ef.val_dir) as n;
   if upd.newat < (select arrivaltime from _arrivaltime_
   where vid = upd.vid) then
     execute 'delete from _queue_arr_time_ where vid = '
     || upd.vid || ';';
     execute 'update _arrivaltime_ set arrivaltime = '
     || upd.newat || ' where vid = '
     || upd.vid || ';';
     insert into _queue_arr_time_
     select at.vid, at.time, at.arrivaltime
     from _arrivaltime_ as at
     where at.vid = upd.vid;
   end if;
end loop;
if n_i = ve then
    drop table _queue_arr_time_;
    return;
end if; end loop;
end; $body$
  language 'plpgsql' volatile strict
  cost 1;
```

B.2.2 Path-selection step—TDSP-GDT

```
create or replace function dr_pathselection(edge_table varchar,vs
bigint, ve bigint)
  returns void as
$body$
declare v_j bigint; v_i bigint; edg record; g_j
numeric(20,2); g_i numeric(20,2); w_gi numeric(20,2); _check
numeric(20,2); id integer;
begin id = 0; _check = 0.99; v_j := ve;
if (select arrivaltime from _arrivaltime_ where vid = ve) > 86400
then
  raise notice 'it was not possible to find a path';
  return;
else while v_j <> vs loop
  for edg in execute 'select e.* from ' || quote_ident(edge_table) || '
  as e inner join _arrivaltime_ as a
  on e.f_node = a.vid where e.t_node = ' || v_j::text
  order by a.arrivaltime
  loop
    v_i := edg.f_node;
    g_i := (select arrivaltime
            from _arrivaltime_ as at
            where at.vid = v_i);
    g_j := (select arrivaltime
            from _arrivaltime_ as at
            where at.vid = v_j);
    w_gi := (select dr_delay_value(ef.geom,g_j)
            from _edgedelay_function_ as ef
            where ef.eid = edg.id and ef.val_dir = edg.val_dir);
    if (g_j / (g_i + w_gi)) > _check then
      v_j := v_i;
      id = id + 1;
      insert into paths values (id, edg.id, edg.val_dir);
      exit;
    end if;
  end loop;
end loop; end if; end; $body$
  language 'plpgsql' volatile strict
  cost 1;
```

B.2.3 Shortest-path request—TDSP-GDT

```
create or replace function dr_2std_shortest_path(edge_table varchar,
node_table varchar, edgedelay_table varchar, vs bigint, ve bigint,
ts double precision, dw integer)
  returns record as
$body$
---node_table is the table with the nodes of the graph;
---edge_table is the table with the edges of the graph;
---edgedelay_table is the table with the edge-delay functions;
---vs start node id;
---ve end node id;
---ts initial time of interval for departure time;
---te final time of interval for departure time;
---dw day of week;
declare traveltime numeric(10,2); distance numeric(10,2); result
record; begin perform dr_initialize(node_table,
edgedelay_table,vs,ve,ts,dw); perform
dr_time_refinement(edge_table,vs,ve,ts,dw); perform
dr_pathselection(edge_table,vs,ve); if (select count(*) from paths)
> 0 then
  traveltime := (select at.arrivaltime - ts
                 from _arrivaltime_ as at
                 where at.vid = ve);
  distance := (select sum(s.meters) from shortest_path as s)/1000;
```

```
else
  distance := 0;
  traveltime := 0;
end if; result := (traveltime, distance); --drop table
_arrivaltime_; drop table _edgedelay_function_; return result; end;
$body$
  language 'plpgsql' volatile strict
  cost 1;
```

B.3 Two-step LTT for TDSP-LTT

B.3.1 Auxiliary tools

```
create or replace function dr_initialtime_to_update
(nodeid bigint, tau_i double precision)
  returns double precision as
$body$
--vid is the node identifier;
--tau_i is the lower bound of the subinterval to be updated,
--determined in the main code;
declare
upd_ts double precision;
wr_te double precision;
begin
wr_te := (select time_e
          from _control_node_subinterval_
          where vid = nodeid);
if wr_te <= tau_i then
  upd_ts := tau_i;
else
  upd_ts := wr_te;
end if;
return upd_ts;
end;
$body$
  language 'plpgsql' immutable strict
  cost 1;

create or replace function dr_finaltime_to_update
(nodeid bigint, tau_prime double precision)
  returns double precision as
$body$
--vid is the node identifier;
--tau_prime is the upper bound of the subinterval to be updated,
--determined in the main code;
declare
upd_te double precision;
wr_te double precision;
begin
wr_te := (select time_e
          from _control_node_subinterval_
          where vid = nodeid);
if wr_te < tau_prime then
  upd_te := tau_prime;
else
  upd_te := 0;
end if;
return upd_te;
end;
$body$
  language 'plpgsql' immutable strict
  cost 1;
```

B.3.2 Time-refinement step—TDSP-LTT

```
create or replace function dr_time_refinement(edge_table character
varying, vs bigint, ve bigint, ts double precision, te double
precision, dw integer)
  returns void as
$body$ -- edge_table is the table that holds the edges of the graph;
--vs is the identifier of source node for the shortest path request;
--ve is the identifier of destination node for the shortest path
request; --ts is the lower bound of absolute time (in seconds along
the day) of an interval in which the shortest path request (least
travel time) is made; --te is the upper bound of absolute time (in
seconds along the day) of an interval in which the shortest path
request (least travel time) is made; --dw is the day of the week (1
- sunday up to 7 - saturday) declare ltt record; -- to fetch rows of
a table in loop inside the algorithm tau_i double precision; --
lower bound of subinterval (absolute time in seconds along the day)
in which the outgoing nodes from the current in iteration will be
updated (and the node in iteration is well-defined) tau_prime double
precision; -- upper bound of subinterval (absolute time in seconds
along the day) in which the outgoing nodes from the current in
iteration will be updated (and the node in iteration is
well-defined) upd_tau_i double precision; upd_tau_prime double
precision; g_ti double precision; -- arrival-time function value
dequeued (the node that will enter in iteration) tau_k double
precision; -- departure time (absolute time in seconds along the
day) of the next node-pair(departure time, arrival-time) in the
queue g_tk double precision; -- arrival-time function value of the
next node-pair(departure time, arrival-time) in the queue geo_ti
geometry; -- arrival-time function of the node in current iteration
delta double precision; -- parameter of the algorithm that together
with g_tk allows to identify the subinterval in which the node in
iteration is well-defined (and then we can update the outgoing nodes
on the basis of that) n_i bigint; -- node identifier of the node in
current iteration n_k bigint; -- node identifier of the next node in
the queue upd arrival_time_function; -- to get the updated
arrival-time function and then update in the table
arrival_time_function countq integer; -- debug stuff _now
timestamp;-- performance check stuff perf timestamp;-- performance
check stuff
begin
tau_i := ts; tau_k := tau_i; --raise notice ’value
of variable after operation =
--%’, tau_i;
while (select count(*)
       from _queue_arr_time_ as q
       ) >= 2 loop
--_now := clock_timestamp();
---dequeue part
countq := (select count(*) from _queue_arr_time_);
raise notice ’countq %’, countq;
select q.vid, q.at_tau, q.tau
  from _queue_arr_time_ as q
  order by q.at_tau
  limit 1 into n_i,g_ti,tau_i;
delete from _queue_arr_time_ where vid=n_i and tau=tau_i; select
q.vid, q.at_tau, q.tau
  from _queue_arr_time_ as q
  where q.vid <> n_i ---and q.active = true
  order by q.at_tau
  limit 1 into n_k,g_tk, tau_k;
raise notice ’n_i =
%’, n_i;
if g_tk is null then
  g_tk := 2592090000;
end if; geo_ti := (select a.geom from arrival_time_function as a
where a.vid = n_i); --perf := clock_timestamp();
--raise notice '%', extract(second from perf - _now);
--raise notice 'tau_i =
 --  %', tau_i;
---subinterval part
--_now := clock_timestamp(); if n_i = vs then
  tau_prime := te::numeric(10,2);
else
  if g_tk > 86400 then
    tau_prime := te;
  else
--_now := clock_timestamp();
  execute 'select min(dr_delay_value(f.tempgeom,'|| g_tk::text || '))
              from ' || quote_ident(edge_table) ||' as e inner join
              _subset_edgedelay_function_ as f on e.id=f.eid
              where e.t_node =' || n_i::text into delta;
--perf := clock_timestamp();
--raise notice '%', extract(second from perf - _now);
   tau_prime := (select dr_max_startingtime_value
   (geo_ti,g_tk + delta,te));
   --  raise notice 'tau_prime =
  --  %', tau_prime;
  end if;
end if; --countq := (select count(*) from _queue_arr_time_);
 -- raise notice 'countq =
 --    %', countq;
delete from _queue_arr_time_ as q where q.vid = n_i and q.tau >=
tau_i and q.tau <= tau_prime; --countq := (select count(*) from
_queue_arr_time_); --  raise notice 'countq =
 --    %', countq;
--_now := clock_timestamp(); if n_i <> vs then
  update _control_node_subinterval_ set time_s = case when time_s = 0
  then tau_i else time_s end, time_e = tau_prime where vid = n_i;
end if;

--perf := clock_timestamp();
--raise notice '%', extract(second from perf - _now);
---update part
if tau_i < tau_prime then for ltt in execute 'select * from
arrival_time_function as at inner join ' || quote_ident(edge_table)
|| ' as e on e.t_node=at.vid where e.f_node=' || n_i::text loop
  upd_tau_i := dr_initialtime_to_update(ltt.t_node,tau_i);
  upd_tau_prime := dr_finaltime_to_update(ltt.t_node,tau_prime);
  --raise notice 'upd_tau_prime = %', upd_tau_prime;
   if upd_tau_prime > 0 then
 -- _now := clock_timestamp();
  select into upd n.vid, st_simplify(
  dr_arrivaltime_function(geo_ti,n.geom,ltt.geom,upd_tau_i,upd_tau_prime),0)
            from (select ltt.t_node as vid, ef.tempgeom as geom
                from _subset_edgedelay_function_ as ef
                where ltt.id=ef.eid and ltt.val_dir=ef.val_dir) as n;
   ---raise notice 'upd.geom %', st_astext(upd.geom);
   --  raise notice '%', st_numpoints(upd.geom);
   update arrival_time_function set geom = st_simplify(upd.geom,0)
   where vid = upd.vid;
--perf := clock_timestamp();
--raise notice 'perf = %', extract(second from perf - _now);
  --raise notice 'upd.vid %',upd.vid;
   --raise notice  'i finished updating!';
---queueing part
--_now := clock_timestamp();
  delete from _queue_arr_time_ where vid=ltt.t_node and tau >= upd_tau_i
  and tau <= upd_tau_prime;
  insert into _queue_arr_time_
  select upd.vid, st_x(st_pointn(upd.geom,n)), st_y(st_pointn(upd.geom,n))
  from generate_series(1,st_numpoints(upd.geom)) as h(n)
  where st_x(st_pointn(upd.geom,n)) >= upd_tau_i
  and st_x(st_pointn(upd.geom,n)) <= upd_tau_prime;
  end if;
--perf := clock_timestamp();
--raise notice '%', extract(second from perf - _now);
end loop;
---final part
--countq := (select count(*) from _queue_arr_time_); --delete from
_queue_arr_time_ as q using _control_node_subinterval_ as c where
q.vid = n_i and q.tau >= c.time_s and q.tau <= c.time_e; --perf :=
clock_timestamp();
--raise notice '%', extract(second from perf - _now);
--_now := clock_timestamp(); --raise notice  'i finished inserting
to queue!'; tau_i := tau_prime; if tau_i >= te then
  if n_i = ve then
    --drop table _queue_arr_time_;
    delete from _queue_arr_time_;
    return;
  end if;
else
  insert into _queue_arr_time_
  select at.vid, tau_prime, g_tk + delta
  from arrival_time_function as at
  where at.vid=n_i;
end if; --perf := clock_timestamp();
--raise notice '%', extract(second from perf - _now);
end if; end loop; end; $body$
  language 'plpgsql' volatile strict
  cost 1;
```

B.3.3 Path-selection step—TDSP-LTT

```
create or replace function dr_pathselection_sg(edge_table varchar,vs
bigint, ve bigint, t_opt double precision)
  returns void as
$body$
declare v_j bigint; v_i bigint; edg record; id integer;
seg_path nldnld_nw; g_j numeric(20,2); g_i numeric(20,2); w_gi
numeric(20,2); _check numeric(20,2);
begin
 _check = 0.99; v_j := ve;
id := 0; while v_j <> vs loop
  for edg in execute 'select * from ' || quote_ident(edge_table) || '
  where t_node = ' || v_j::text
  loop
  raise notice 'edg.f_node=%', edg.f_node;
    v_i := edg.f_node;
    raise notice 'v_i=%', v_i;
    g_i := (select dr_delay_value(at.geom, t_opt)
            from arrival_time_function as at
            where at.vid = v_i);
            raise notice 'g_i=%', g_i;
    g_j := (select dr_delay_value(at.geom, t_opt)
            from arrival_time_function as at
            where at.vid = v_j);
            raise notice 'g_j=%', g_j;
    w_gi := (select dr_delay_value(ef.tempgeom,g_j)
            from _subset_edgedelay_function_ as ef
            where ef.eid = edg.id and ef.val_dir = edg.val_dir);
            raise notice 'w_gi=%', w_gi;
            raise notice 'g_j-g_i=%', g_j-g_i;
    if (g_j / (g_i + w_gi)) > _check then
      v_j := v_i;
      id = id + 1;
      raise notice 'v_i=%', v_j;
      insert into paths values (id, edg.id, edg.val_dir);
```

```
               exit;
  end if;
 end loop;
end loop; end; $body$
 language 'plpgsql' volatile strict
 cost 1;
```

B.3.4 Shortest-path request—TDSP-LTT

```
create or replace function dr_2std_shortest_path_sg(edge_table
varchar, node_table varchar, edgedelay_table varchar, vs bigint, ve
bigint, ts double precision, te double precision, dw integer)
  returns record as
$body$
declare traveltime numeric(10,2); distance numeric(10,2);
t_star numeric(10,2); result record;
begin
perform
dr_initialize(node_table,edgedelay_table, vs,ve,ts,te,dw);
perform
dr_time_refinement(edge_table,vs,ve,ts,te,dw);
t_star  := (select
dr_optimal_startingtime(geom)
           from arrival_time_function
           where vid = ve);
perform dr_pathselection(edge_table,vs,ve,t_star); if (select
count(*) from paths) > 0 then
  traveltime := (select dr_delay_value(geom,t_star) - ts
                 from arrival_time_function
                 where vid = ve);
  distance := (select sum(s.meters) from shortest_path_sg as s)/1000;
else
  distance := 0;
  traveltime := 0;
end if; result := (t_star,traveltime, distance); drop table
_subset_edgedelay_function_; drop table _control_node_subinterval_;
return result; end; $body$
  language 'plpgsql' volatile strict
  cost 1;
```

Appendix C

Optimization procedures

C.1 Sequence of commands for the trivial graph simplification

```
create table subset_dissolved_edges (id integer, network_id bigint
not null, constraint subset_dissolved_edges_pk primary key
(network_id));

alter table subset_dissolved_edges add column seq serial;

create table subset_nldnld_nw (like nldnld_nw including
constraints); alter table subset_nldnld_nw add constraint
subset_nldnld_nw_pk primary key (gid)

insert into subset_nldnld_nw select n.* from nldnld_nw as n where
(n.frc = 1 or n.frc = 0) and
st_within(n.geom,st_setsrid(st_makebox3d(
st_makepoint(4.2378,51.7516),st_makepoint(5.68852,52.62677)),4326));

create table subset_nldnld_jc (like nldnld_jc including
constraints); alter table subset_nldnld_jc add constraint
subset_nldnld_jc_pk primary key (gid)

create table subset_node (vid bigint not null, constraint
subset_node_pk primary key (vid));

insert into subset_node select q.nid from (select f_jnctid as nid
from subset_nldnld_nw union all select t_jnctid as nid from
subset_nldnld_nw) as q group by q.nid having count(*) = 1 or
count(*) > 2;

select distinct r.* from subset_nldnld_nw as r, (select j.geom from
subset_nldnld_jc as j where j.id not in (select * from subset_node
as n)) as v where st_intersects(v.geom,r.geom);

insert into subset_nldnld_jc select j.* from nldnld_jc as j inner
join subset_node as n on j.id = n.vid

create or replace function dr_dissolve_edges() returns void as
$$
declare v_j bigint; seg_id bigint; e_id integer; edg record; begin
e_id := 0; for edg in select * from subset_edge_majroad where f_node
in (select vid from subset_node) loop
  e_id := e_id + 1;
  v_j := edg.t_node;
  raise notice 'v_j %', v_j;
  insert into subset_dissolved_edges
  (network_id, id) values (edg.id, e_id);
  while not v_j in (select vid from subset_node) loop
    seg_id := (select id from subset_edge_majroad
    where f_node = v_j);
    raise notice 'seg_id %', seg_id;
    raise notice 'e_id %', e_id;
    insert into subset_dissolved_edges (network_id, id)
    values (seg_id, e_id);
    v_j := (select t_node from subset_edge_majroad
    where id = seg_id);
    raise notice 'v_j %', v_j;
  end loop;
end loop; end;
$$
language 'plpgsql' volatile strict cost 1;

select dr_dissolve_edges();

create table subset_edge_majroad (like edge_majroad including
constraints);

insert into subset_edge_majroad select e.* from edge_majroad as e
inner join subset_nldnld_nw as r on e.id = r.id;

create table subset_edge_majroad_simplified (like edge_majroad
including constraints); alter table subset_edge_majroad_simplified
add constraint subset_edge_majroad_simplified_pk primary key
(id,val_dir);

insert into subset_edge_majroad_simplified
(id,f_node,val_dir,length) select e.id, m.f_node, m.val_dir,
m.length from subset_edge_majroad as m inner join (select d.* from
(select min(seq) from subset_dissolved_edges group by id) as o inner
join subset_dissolved_edges as d on o.min=d.seq) as e on
e.network_id = m.id;

update subset_edge_majroad_simplified as s set t_node = l.t_node
from (select e.id, m.t_node from subset_edge_majroad as m inner join
(select d.* from (select max(seq) from subset_dissolved_edges group
by id) as o inner join subset_dissolved_edges as d on o.max=d.seq)
as e on e.network_id = m.id) as l where l.id = s.id;

create table edge_delay_function (like _edge_delay_function_
including constraints); alter table edge_delay_function add
constraint egde_delay_function_pk primary key (eid,val_dir);

insert into edge_delay_function select n.id, n.val_dir,
st_snaptogrid(st_scale(st_simplify(s.geominv,0),
1,n.length*3.6/n.spfreeflow),0,0.01)
from (select edg.id, edg.val_dir, edg.length, case when p.spfreeflow
> 0 then p.spfreeflow else p.spweekday end as spfreeflow,
            case when p.profile_4 = 0 then 1 else p.profile_4
            end as profile
       from nldnld_hsnp as p, subset_edge_majroad as edg
       where edg.id=p.network_id and edg.val_dir=p.val_dir) as n,
       speed_profile_function as s
where s.pid=n.profile;
```

C.2 Generating aggregated edge-delay functions

```
create or replace function dr_aggregate_edgedelay(dw integer)
returns void as
```

```
$$
declare c integer; e record; geos geometry; geo geometry;
param_query text; i_plus integer; begin for e in select distinct id
from subset_edge_majroad_simplified loop
  c := (select count(*) from subset_dissolved_edges where id = e.id);
  RAISE NOTICE 'e.id %', e.id;
  if (c-1) > 0 then
      param_query := 'select st_collect(f.geom_' || dw::text || ')
             from edge_delay_function as f
             inner join subset_dissolved_edges as d
             on f.eid = d.network_id
             where d.id = ' || e.id::text;
      execute param_query into geos;
      RAISE NOTICE ' geos %', st_astext(geos);
      i_plus := 1;
      geo := (select st_geometryn(geos,i_plus));
      RAISE NOTICE ' geo %', st_astext(geo);
      for i in 1..(c-1) loop
        i_plus := i + 1;
        geo := (select st_snaptogrid(st_simplify(
        dr_aggregated_edgedelay(geo,
        st_geometryn(geos,i_plus)),0),0,0.01) );
      end loop;
      RAISE NOTICE ' geo %', st_astext(geo);
      update subset_edgedelay_function set tempgeom = geo
      where eid=e.id;
  else
      param_query := 'select f.geom_' || dw::text || '
             from edge_delay_function as f
             inner join subset_dissolved_edges as d
             on f.eid = d.network_id
             where d.id = ' || e.id::text;
      execute param_query into geo;
      RAISE NOTICE ' geo %', st_astext(geo);
      update subset_edgedelay_function set tempgeom = geo
      where eid=e.id;
  end if;
end loop; end;
$$
language 'plpgsql' volatile strict cost 1;

create table subset_edgedelay_function ( eid integer not null,
  val_dir smallint not null,
  constraint  subset_edgedelay_function_pk primary key (eid,val_dir)
) select AddGeometryColumn
('dynamic_routing','subset_edgedelay_function',
'tempgeom','-1','LINESTRING',2);


insert into subset_edgedelay_function select id, val_dir from
subset_edge_majroad_simplified

select dr_aggregate_edgedelay(1); update subset_edgedelay_function
set geom_1 = tempgeom; select dr_aggregate_edgedelay(2); update
subset_edgedelay_function set geom_2 = tempgeom; select
dr_aggregate_edgedelay(3); update subset_edgedelay_function set
geom_3 = tempgeom; select dr_aggregate_edgedelay(4); update
subset_edgedelay_function set geom_4 = tempgeom; select
dr_aggregate_edgedelay(5); update subset_edgedelay_function set
geom_5 = tempgeom; select dr_aggregate_edgedelay(6); update
subset_edgedelay_function set geom_6 = tempgeom; select
dr_aggregate_edgedelay(7); update subset_edgedelay_function set
geom_7 = tempgeom;
```

C.2.1 Aggregated edge-delay function algorithm

```
create or replace function dr_aggregated_edgedelay(geometry,
geometry) returns geometry as $body$
---here as we know that between after 21:30 and 5:00 of the next
---day the relative speed is 100%,
---we can assume the last delay value of the day as the one to be
---added after the domain
---of the second function would be finished (went around to the next day)
select st_linefrommultipoint(st_collect(st_makepoint(q.t,q.delay)))
from (select p.t, case when (p.t + dr_delay_value($1,p.t)) > 86400
then
            dr_delay_value($1,p.t) + dr_delay_value($2,0)
            else dr_delay_value($1,p.t) + dr_delay_value($2, p.t
            + dr_delay_value($1,p.t)) end
            as delay
from (select st_x(st_pointn($1,n)) as t from
generate_series(1,st_numpoints($1)) as h(n) union select
st_x(st_pointn($2,n)) from generate_series(1,st_numpoints($2)) as
h(n)) as p order by p.t) as q $body$ language 'sql' immutable strict
cost 1;
```

C.3 Sequence of commands for the dense subgraph simplification

```
create table cluster_pol (id integer not null, constraint
cluster_pol_pk primary key (id));

select addgeometrycolumn
('dynamic_routing','cluster_pol','geom','4326','POLYGON',2);

---polygons created in QuantumGIS.
```

C.4 Identify entry nodes and exit nodes

```
create or replace function dr_entry_nodes(edge_table character
varying)
  returns setof bigint as
$body$
---edge_table is the table that holds the edges inside a tile;
declare edg record; c integer; begin for edg in execute 'select *
from '|| quote_ident(edge_table) loop
  execute 'select  count(*) from ' || quote_ident(edge_table) || '
  where t_node = ' || edg.f_node into c;
  if c = 0 then
     return next edg.f_node;
  end if;
end loop; end; $body$
  language 'plpgsql' immutable strict
  cost 1
  rows 1000;

create or replace function dr_exit_nodes(edge_table character
varying)
  returns setof bigint as
$body$
---edge_table is the table that holds the edges inside a tile;
```

```
declare edg record; c integer; begin for edg in execute 'select *
from '|| quote_ident(edge_table) loop
  execute 'select  count(*)::text from ' || quote_ident(edge_table)
  || ' where f_node = ' || edg.t_node into c;
  if c = 0 then
     return next edg.t_node;
  end if;
end loop; end; $body$
  language 'plpgsql' immutable strict
  cost 1
  rows 1000;
```

C.4.1 Precomputation of least-delay functions and graph simplification

This code performs all the clusters at once.

```
create or replace function dr_precomputation_cluster( nw_table
varchar, edge_table varchar, collap_edge_table varchar,
edgedelay_table varchar, dis_edge_table varchar, collap_edgedelay
varchar, cluster_table varchar) returns void as
$$
---nw_table is the table with the (geometry) network (2-core)
---collap_edge_table is the table with the collapsed egdes (2-core)
---dis_edge_table is the link table collap_edge_table - edge_table
---edge_table is the table with the original edges
---edgedelay_table is the table with the edge-delay functions
---collap_edgedelay has the edge-delay functions of the
---collapsed edges (2-core)
---cluster_table is the table with the zones to be simplified
declare clu record; begin create table tile_edge (
  id bigint not null,
  val_dir smallint not null,
  f_node bigint,
  t_node bigint,
  length double precision,
  constraint tile_edge_pk1 primary key (id, val_dir)
); create table control_tile_edge (
  id bigint not null,
  val_dir smallint not null,
  constraint control_tile_edge_pk1 primary key (id,val_dir)
); create unique index tile_edge_idx
  on tile_edge
  using btree
  (id, val_dir);

create index tile_egde_tnode_idx
  on tile_edge
  using btree
  (t_node);

create table tile_entrynode (
  vid bigint not null,
  constraint tile_entrynode_pk1 primary key (vid)
);

create table tile_exitnode (
  vid bigint not null,
  constraint tile_exitnode_pk1 primary key (vid)
); create table tile_vgraph (
  f_node bigint,
  t_node bigint,
  constraint tile_vgraph_pk1 primary key (f_node,t_node)
); create table tile_edgedelay (
  eid bigint not null,
  val_dir smallint not null,
  constraint tile_egdedelay_pk primary key (eid, val_dir));
perform addgeometrycolumn
('dynamic_routing','tile_edgedelay','geom','-1','linestring',2);
perform addgeometrycolumn
('dynamic_routing','tile_vgraph','geom','-1','linestring',2);
create
table simplified_edge (
  id bigint not null,
  val_dir smallint not null,
  f_node bigint,
  t_node bigint,
  length double precision,
  constraint simplified_edge_pk1 primary key (id, val_dir)
);

create unique index simplified_edge_idx
  on simplified_edge
  using btree
  (id, val_dir);

create index simplified_egde_tnode_idx
  on simplified_edge
  using btree
  (t_node);
create table simplified_edgedelay (
  eid bigint not null,
  val_dir integer not null,
  constraint simplified_egdedelay_pk primary key (eid, val_dir));
perform addgeometrycolumn
('dynamic_routing','simplified_edgedelay',
'geom_2','-1','linestring',2);
create table simplified_node (
  vid bigint not null,
  constraint nonsimplified_node_pk1 primary key (vid)
); for clu in execute 'select * from '
|| quote_ident(cluster_table)
loop
  execute
  ---fetch 'mesh'of the graph inside the defined cluster
  'insert into tile_edge
  select e.id, e.val_dir, e.f_node, e.t_node, e.length
  from ' || quote_ident(edge_table) || ' as e,
  (select d.network_id as id
  from ' || quote_ident(nw_table) ||
  ' as e inner join ' || quote_ident(dis_edge_table ) ||
  ' as d on e.id = d.id
  where st_within(geom,st_setsrid(st_geomfromtext(''' ||
 st_astext(clu.geom)::text || '''),4326))) as g
  where e.id = g.id';
  ---getting the edges to control the rest of the graph
  insert into control_tile_edge
  select id, val_dir
  from tile_edge;
  ---identify entry nodes
  insert into tile_entrynode
  select distinct *
  from dr_entry_nodes('tile_edge');
  ---identify exit nodes
  insert into tile_exitnode
  select distinct *
  from dr_exit_nodes('tile_edge');
  ---fetch edge-delay functions
  execute
  'insert into tile_edgedelay
  select f.eid, f.val_dir, f.geom_2
  from tile_edge as e inner join '|| quote_ident(edgedelay_table)
  || ' as f on e.id = f.eid and e.val_dir = f.val_dir';
  ---creating all-pairs combinations between entry nodes and other nodes
  insert into tile_vgraph
  select n.vid as f_node, v.vid as t_node
  from tile_entrynode as n,
  (select t_node as vid
  from tile_edge
  where t_node not in (select vid from tile_entrynode)
  union
  select f_node
  from tile_edge
  where f_node not in (select vid from tile_entrynode)) as v;
  ---initialize least-delay functions
  update tile_vgraph set geom = dr_others_g_function(0,86400);
  ---compute least-delay functions
  perform dr_least_delay_function
  ('tile_vgraph', 'tile_edge', 'tile_edgedelay',
  'tile_entrynode');
  ---creating the simplified edges for entry x exit nodes
  insert into simplified_edge
  select (substring(f_node::text from 9) ||
  substring(t_node::text from 9))::bigint,
  1, f_node, t_node
  from tile_vgraph
  where t_node in (select vid from tile_exitnode) and not geom =
  dr_others_g_function(0,86400);
  ---inserting the precomputed least-delay functions
  insert into simplified_edgedelay
  select (substring(f_node::text from 9) ||
  substring(t_node::text from 9))::bigint,
  1, geom
  from tile_vgraph
  where t_node in (select vid from tile_exitnode) and not geom =
  dr_others_g_function(0,86400);
  ---clean-up tables for the next cluster
  delete from tile_edge;
  delete from tile_entrynode;
  delete from tile_exitnode;
  delete from tile_edgedelay;
  delete from tile_vgraph;
  raise notice 'going for other cluster...';
end loop;
---getting the rest of the graph
execute 'insert into simplified_edge select * from ' ||
quote_ident(collap_edge_table) || ' where id not in (select distinct
d.id from ' || quote_ident(dis_edge_table) || ' as d inner join
control_tile_edge as e on d.network_id = e.id)'; insert into
simplified_node select t_node from simplified_edge union select
f_node from simplified_edge; execute 'insert into
simplified_edgedelay select distinct e.eid, e.val_dir, e.geom_2 from
'  || quote_ident(dis_edge_table) || ' as d inner join ' ||
quote_ident(collap_edgedelay) ||' as e on d.id=e.eid where
d.network_id not in (select id from control_tile_edge)'; drop table
tile_edge; drop table tile_entrynode; drop table tile_exitnode; drop
table tile_edgedelay; drop table tile_vgraph; drop table
control_tile_edge; end;
$$
language 'plpgsql' volatile strict cost 1;
```

This sequence of code is to perform each clusters per time.

```
create table simplified_edge (
  id bigint not null,
  val_dir smallint not null,
  f_node bigint,
  t_node bigint,
  length double precision,
  constraint simplified_edge_pk1 primary key (id, val_dir)
);

create unique index simplified_edge_idx
  on simplified_edge
  using btree
  (id, val_dir);

create index simplified_egde_tnode_idx
  on simplified_edge
  using btree
  (t_node);
create table simplified_edgedelay (
  eid bigint not null,
  val_dir integer not null,
  constraint simplified_egdedelay_pk primary key (eid, val_dir));
perform addgeometrycolumn
('dynamic_routing','simplified_edgedelay','geom_2',
'-1','linestring',2);
create table simplified_node (
  vid bigint not null,
  constraint nonsimplified_node_pk1 primary key (vid)
);

create table control_tile_edge (
  id bigint not null,
  val_dir smallint not null,
  constraint control_tile_edge_pk1 primary key (id,val_dir)
);

create or replace function dr_precomputation_percluster( clus
geometry, nw_table character varying, edge_table character varying,
collap_edge_table character varying, edgedelay_table character
varying,
 dis_edge_table character varying, collap_edgedelay character varying)
  returns void as
$body$
---nw_table is the table with the (geometry) network (2-core)
---collap_edge_table is the table with the collapsed egdes (2-core)
---dis_edge_table is the link table collap_edge_table - edge_table
---edge_table is the table with the original edges
---edgedelay_table is the table with the edge-delay functions
---collap_edgedelay has the edge-delay functions of the
---collapsed edges (2-core)
---clus is the dense subgraph zone to be simplified
begin create table tile_edge (
  id bigint not null,
  val_dir smallint not null,
  f_node bigint,
  t_node bigint,
  length double precision,
  constraint tile_edge_pk1 primary key (id, val_dir)
);

create unique index tile_edge_idx
  on tile_edge
  using btree
  (id, val_dir);

create index tile_egde_tnode_idx
  on tile_edge
  using btree
  (t_node);
```

```
create table tile_entrynode (
  vid bigint not null,
  constraint tile_entrynode_pk1 primary key (vid)
); create table tile_exitnode (
  vid bigint not null,
  constraint tile_exitnode_pk1 primary key (vid)
); create table tile_vgraph (
  f_node bigint,
  t_node bigint,
  constraint tile_vgraph_pk1 primary key (f_node,t_node)
); create table tile_edgedelay (
  eid bigint not null,
  val_dir smallint not null,
  constraint tile_egdedelay_pk primary key (eid, val_dir));
perform addgeometrycolumn
('dynamic_routing','tile_edgedelay','geom','-1','linestring',2);
perform addgeometrycolumn
('dynamic_routing','tile_vgraph','geom','-1','linestring',2);

execute
  ---fetch 'mesh'of the graph inside the defined cluster
  'insert into tile_edge
  select e.id, e.val_dir, e.f_node, e.t_node, e.length
  from ' || quote_ident(edge_table) || ' as e,
  (select d.network_id as id
  from ' || quote_ident(nw_table) ||
  ' as e inner join ' || quote_ident(dis_edge_table ) ||
  ' as d on e.id = d.id
  where st_within(geom,st_setsrid(st_geomfromtext(''' ||
 st_astext(clus)::text || '''),4326))) as g
  where e.id = g.id';
  ---getting the edges to control the rest of the graph
  insert into control_tile_edge
  select id, val_dir
  from tile_edge;
  ---identify entry nodes
  insert into tile_entrynode
  select distinct *
  from dr_entry_nodes('tile_edge');
  ---identify exit nodes
  insert into tile_exitnode
  select distinct *
  from dr_exit_nodes('tile_edge');
  ---fetch edge-delay functions
  execute
  'insert into tile_edgedelay
  select f.eid, f.val_dir, f.geom_2
  from tile_edge as e inner join '|| quote_ident(edgedelay_table) || ' as f
  on e.id = f.eid and e.val_dir = f.val_dir';
  ---creating all-pairs combinations between entry nodes and other nodes
  insert into tile_vgraph
  select n.vid as f_node, v.vid as t_node
  from tile_entrynode as n,
  (select t_node as vid
  from tile_edge
  where t_node not in (select vid from tile_entrynode)
  union
  select f_node
  from tile_edge
  where f_node not in (select vid from tile_entrynode)) as v;
  ---initialize least-delay functions
  update tile_vgraph set geom = dr_others_g_function(0,86400);
  ---compute least-delay functions
  perform dr_least_delay_function('tile_vgraph', 'tile_edge',
  'tile_edgedelay',
  'tile_entrynode');
  ---creating the simplified edges for entry x exit nodes
  insert into simplified_edge
  select (substring(f_node::text from 9) ||
  substring(t_node::text from 9))::bigint,
  1, f_node, t_node
  from tile_vgraph
  where t_node in (select vid from tile_exitnode) and not geom =
  dr_others_g_function(0,86400);
  ---inserting precomputed least-delay functions for simplified edges
  insert into simplified_edgedelay
  select (substring(f_node::text from 9) ||
  substring(t_node::text from 9))::bigint,
  1, geom
  from tile_vgraph
  where t_node in (select vid from tile_exitnode) and not geom =
  dr_others_g_function(0,86400);
  ---clean-up tables for the next cluster
  delete from tile_edge;
  delete from tile_entrynode;
  delete from tile_exitnode;
  delete from tile_edgedelay;
  delete from tile_vgraph;
drop table tile_edge; drop table tile_entrynode; drop table
tile_exitnode; drop table tile_edgedelay; drop table tile_vgraph;
end; $body$
  language 'plpgsql' volatile strict
  cost 1;

  create or replace function dr_get_rest_graph(
  collap_edge_table character varying,
  dis_edge_table character varying,
  collap_edgedelay character varying)
  returns void as
$body$
---collap_edge_table is the table with the collapsed egdes (2-core)
---dis_edge_table is the link table collap_edge_table - edge_table
---collap_edgedelay has the edge-delay functions
---of the collapsed edges (2-core)
begin execute 'insert into simplified_edge select * from ' ||
quote_ident(collap_edge_table) || ' where id not in
(select distinct d.id from ' || quote_ident(dis_edge_table)
|| ' as d inner join
control_tile_edge as e on d.network_id = e.id)';
insert into simplified_node
select t_node from simplified_edge union select
f_node from simplified_edge; execute 'insert into
simplified_edgedelay select distinct e.eid, e.val_dir, e.geom_2 from
'  || quote_ident(dis_edge_table) || ' as d inner join ' ||
quote_ident(collap_edgedelay) ||' as e on d.id=e.eid where
d.network_id not in (select id from control_tile_edge)'; end;
$body$
  language 'plpgsql' volatile strict
  cost 1;
```

C.4.2 Average performance tests

Performance check for varying distance and fixed time interval for departure.

```
create or replace function dr_opt_performance_test
(edge_table character varying, node_table character varying,
edgedelay_table character varying, vs bigint, ts double precision,
te double precision, dw integer)
  returns setof interval as
```

```
$body$
---edge_table is the table that holds the simplified edges;
---node_table is the table that holds the simplified nodes;
---edgedelay_table is the table that holds the least-delay functions;
---vs is the start node for the search;
---ts is the initial time of the interval of departure for refinement;
---te is the final time of the interval of departure for refinement;
---dw is the day of the week;
declare
result interval;
initime timestamp;
finaltime timestamp;
n record;
begin
for n in execute 'select * from ' || quote_ident(node_table) ||
' where vid <> ' || vs::text || ' limit 20' loop
  initime := clock_timestamp();
  perform dr_initialize(node_table,edgedelay_table, vs,n.vid,ts,te,dw);
  perform dr_time_refinement(edge_table,vs,n.vid,ts,te,dw);
  finaltime := clock_timestamp();
  delete from arrival_time_function;
  drop table _control_node_subinterval_;
  drop table _subset_edgedelay_function_;
  drop table _queue_arr_time_;
  result := (select finaltime - initime);
  raise notice 'going for the next round';
  return next result;
end loop;
end;
$body$
  language 'plpgsql' volatile strict
  cost 1
  rows 1000;
```

Performance check for fixed distance and varying time interval for departure.

```
create or replace function dr_opt_performance_time_test
(edge_table character varying, node_table character varying,
edgedelay_table character varying, vs bigint, ve bigint, dw integer)
  returns setof interval as
$body$
---edge_table is the table that holds the simplified edges;
---node_table is the table that holds the simplified nodes;
---edgedelay_table is the table that holds the least-delay functions;
---vs is the start node for the search;
---ve is the target node for the search;
---dw is the day of the week;
declare
result interval;
initime timestamp;
finaltime timestamp;
te double precision;
r double precision;
n integer;
begin
n = 0;
while n < 20 loop
  r := (select  random());
  te := (select case when r < 0.8333333 then 29400 else 28800 + r*7200 end);
  initime := clock_timestamp();
  perform dr_initialize(node_table,edgedelay_table, vs,ve,28800,te,dw);
  perform dr_time_refinement(edge_table,vs,ve,28800,te,dw);
  finaltime := clock_timestamp();
```

```
    delete from arrival_time_function;
    drop table _control_node_subinterval_;
    drop table _subset_edgedelay_function_;
    drop table _queue_arr_time_;
    result := (select finaltime - initime);
    raise notice 'going for the next round';
    return next result;
    n := n + 1;
  end loop;
  end;
  $body$
    language 'plpgsql' volatile strict
    cost 1
    rows 1000;
```