Point-to-point shortest paths on dynamic time-dependent road networks

Giacomo Nannicini

ÉCOLE POLYTECHNIQUE
ParisTech

Point-to-Point Shortest Paths on Dynamic Time-Dependent Road Networks

Thèse présentée pour obtenir le grade de
DOCTEUR DE L'ECOLE POLYTECHNIQUE

par

Giacomo Nannicini

Soutenue le 18 juin 2009 devant le jury composé de:

Dorothea Wagner	Universität Karlsruhe, Karlsruhe	Rapporteur
Roberto Wolfler-Calvo	Université Paris Nord, Paris	Rapporteur
Gilles Barbier	DisMoiOù, Paris	
Philippe Goudal	Mediamobile, Ivry sur Seine	
Frank Nielsen	Ecole Polytechnique, Palaiseau	
Leo Liberti	Ecole Polytechnique, Palaiseau	Directeur de thèse
Philippe Baptiste	Ecole Polytechnique, Palaiseau	Co-directeur de thèse
Daniel Krob	Ecole Polytechnique, Palaiseau	Co-directeur de thèse

Abstract

The computation of point-to-point shortest paths on time-dependent road networks has many practical applications which are interesting from an industrial point of view. Typically, users are interested in the path leading to their destination which has the smallest travel time among all possible paths; it is natural to model the shortest paths problem on a time-dependent graph, where the arc weights are travel times that depend on the time of day at which the arc is traversed. We study both fully combinatorial methods and mathematical formulation based methods. From a combinatorial point of view, if we impose some restrictions on the arc weights, the problem can be solved in polynomial time with the well known Dijkstra's algorithm. However, applying Dijkstra's algorithm on a graph with several millions of vertices and arcs, such as a continental road network, may require several seconds of CPU time. This is not acceptable for real-time industrial applications; therefore, the need for speedup techniques arises. Bidirectional search is a standard technique to speed up computations on static (i.e. non time-dependent) graphs; however, as the arrival time at the destination is unknown, the cost of time-dependent arcs around the target node cannot be evaluated, thus bidirectional search cannot be directly applied on time-dependent networks. We propose an algorithm based on an asymmetric bidirectional search, which allows the extension to the time-dependent case of hierarchical speedup techniques, well known for static graphs. Our method deals efficiently with dynamic scenarios where arcs weights can change, so that we can take into account real-time and forecast traffic information as soon as it becomes available. We achieve average query times for *time-dependent* shortest paths computations that were previously only possible on dynamic graphs with *static* arc costs. We discuss the integration of our algorithm with an existing real-world industrial application. For general arc weight functions, the problem is not polynomially solvable; we propose a mathematical programming formulation which is a Mixed-Integer Linear Program (MILP) if the time-dependent arc weights are linear or piecewise linear functions, whereas it is a Mixed-Integer Nonlinear Program (MINLP) if the arc weights are nonlinear functions. We study efficient algorithms for both classes of problems, and test them on benchmark instances taken from the literature, as well as shortest paths instances. We propose new branching strategies within the context of a Branch-and-Bound algorithm for MILPs. Computational experiments show that, by generating good branching decisions, we enumerate on average half the nodes enumerated by traditional strategies. Our approach is also competitive in terms of total computational time. Finally, we present a general-purpose heuristic for MINLPs based on Variable Neighbourhood Search, Local Branching, Sequential Quadratic Programming and Branch-and-Bound. Experiments show the reliability of our heuristic with respect to methods proposed in the literature.

Contents

Chapter 1

Introduction

1.1 Motivation

Route planners and associated features are increasingly popular among web users: several web sites provide easy-to-use interfaces that allow users to select a starting and a destination point on a map, and a path between the two points satisfying one or more criteria is computed. Possible criteria are, for example: minimize travel time, total path length or estimated travel cost. Similar capabilities can be found in GPS devices; as these usually have a limited amount of memory and CPU power, several devices now use different kinds of wireless connections in order to query a web service, which computes the desired path using more sophisticated algorithms than those available on the portable device.

Users are typically interested in the *fastest* path to reach their destination, i.e. the shortest path in terms of travel time. However, usually only static information is taken into account when computing this kind of shortest paths, while it is well known that the travel time over a road segment depends on its congestion level, which in turn is dependent on the time instant at which the road segment is traversed. This implicitly requires complete knowledge of both real-time and forecast traffic information over the whole road network, so that we are able to compute the traversal time of a road segment for each time instant in the future. This assumption is obviously unrealistic; nevertheless, several statistical models exist which are able to predict to a certain degree of accuracy the evolution of traffic. This kind of analysis is made possible by traffic sensors (electromagnetic loops, cams, etc.) that are positioned at strategic places of the road network and constantly monitor the traffic situation, providing both high-level information such as the congestion level of a highway and low-level information such as the travel time in seconds over a particular road segment. Using a large database of historical traffic information and statistical analysis tools we can compute *speed profiles* for the different road segments, i.e. cost functions that associate the most probable travel speed (and thus travel time)

over a road segment with the time instant at which the segment is traversed. Typically there will be several classes of these speed profiles, e.g. one class of profiles for weekdays and another one for holidays. A road network such that the travel time over a road segment depends on the time instant at which the segment is traversed is called *time-dependent*. One practical problem arises: as road networks may be very large, traffic sensors cannot cover all road segments. In real-world scenarios, only a small part of the road network is constantly monitored, while the remaining part is not covered by sensors and, as a consequence, by speed profiles. However, the monitored part of the road network corresponds to the most important road segments, e.g. motorways and highways. For long distance paths, the traffic congestion status of these segments is the most important for determining the total travel time, and is also the most significant from a user's point of view: it is reasonable to assume that a car driver which asks for the fastest path to reach the destination wants to avoid traffic jams on high importance roads, which have a large influence on the total travel time, while congestions at local level near the departure or the destination point are less important, as well as more difficult (if not impossible) to foresee. Thus, in a realistic situation only a part of the road network is provided with real-time and forecast traffic information, while the remaining part is associated with static travel times.

This scenario is further complicated by the fact that the speed profiles may not be the most accurate traffic information available. Indeed, it is clear that real-time information, as detected by the traffic sensors, gives the best estimation of travel times for the time instant at which it is gathered. Moreover, several predictive models for short and mid-term traffic forecasting exist, which are beyond the scope of this work and will not be discussed here; these models are based on the real-time information and capitalize on the temporal and spatial locality of traffic jams, so that they are able to predict congestions with a larger degree of accuracy with respect to speed profiles, which only take into account historical data. In the end, the historical speed profiles are not the only source of traffic information: they provide a good estimation of long term traffic dynamics, but for short and mid-term forecasting more accurate dynamic data is available. Therefore, the cost functions that associate travel times to road segments and the time at which the segment is traversed should ideally be *dynamic*, i.e. they should be based on historical speed profiles, but they should be frequently updated in order to take into account both real-time traffic information and short and mid-term traffic forecastings. In the following we will assume that the time required for each shortest path computation is much shorter than the time interval at which real-time traffic information (and thus traffic forecastings) is updated, so that computations can always be carried out before the cost functions are modified. This is realistic in industrial applications, since a shortest path should be computed very quickly (no more than a second), whereas traffic information is typically updated every few minutes.

Under reasonable assumptions, the problem of finding the shortest path in terms of travel time on a time-dependent road network is theoretically solved in polynomial time by Dijkstra's algorithm (see Section 1.2.1 and Section 1.4.2). However, an application of Dijkstra's algorithm over a continental sized road network may require several seconds of CPU time, and in several real-world scenarios this may be too long. For instance, consider the web service scenario: if we assume that there may be several shortest path queries per second, then each shortest path computation should take no more than a few milliseconds. This situation also arises in the case of GPS devices: real-time and forecast traffic information may be difficult to deliver to limited capabilities devices for several reasons (bandwidth, secrecy, etc.), so that the most efficient choice is gathering the traffic information on a server machine with large computational power, which should then quickly provide answers to shortest path queries to all connected devices. This motivates our need for speedup techniques. It is easy to develop heuristic strategies, e.g. for long distance paths we can restrict the search to motorways after a few kilometers away from the starting point, thus neglecting all less important roads. However, both from a theoretical and a practical point of view we are more interested in exact methods, or at least methods with a (small) approximation guarantee. While in other shortest paths applications only exact solutions may be interesting, a small approximation factor is practically acceptable when dealing with road networks, since the input data (i.e. travel times) is affected by measurement errors anyway, and traffic forecasts may fail to be exact.

In the general case, i.e. without restrictions on the cost functions, the time-dependent shortest path problem is NP-hard (see Section 1.2.1). For very large networks, there is no hope of solving it to optimality within a short time; therefore, for real-time applications we are more interested in a restriction of the problem which is polynomially solvable. However, the study of the general case finds application as a mean to verify that the solutions to the polynomially solvable restriction of the problem are meaningful for the network users, even when the restrictions are lifted. We model the time-dependent shortest path problem in a general network through a mathematical program. The greatest advantage of employing a mathematical program is the flexibility of the resulting model: we can choose arbitrary cost function, and easily add complicating constraints that would be difficult to satisfy with a Dijkstra-like approach. For instance, taking into account prohibited turnings is straightforward within the mathematical programming formulation that we propose. This program is a mixed-integer linear program or a mixed-integer nonlinear program, depending on the functions which model the travelling time over the arcs of the network. Instead of searching for specialized algorithms to solve the time-dependent shortest path problem in the general case, we study general-purpose algorithms for mixed-integer linear programs and mixed-integer nonlinear programs. This allows us to improve the performance with respect to the literature of existing

algorithms that solve very large classes of problems, one of which is the routing problem that is the specific subject of this thesis.

1.2 Definitions and Notation

Consider an interval $\mathcal{T} = [0, P] \subset \mathbb{R}$ and a function space $\mathbb{F}$ of positive functions $f : \mathbb{R}^+ \to \mathbb{R}^+$ with the property that $\forall \tau > P \; f(\tau) = f(\tau - kP)$, where $k = \max\{k \in \mathbb{N} | \tau - kP \in \mathcal{T}\}$. This implies $f(\tau + P) = f(\tau) \; \forall \tau \in \mathcal{T}$; in other words, f is periodic of period P. We additionally require that $f(x) + x \leq f(y) + y \; \forall f \in \mathbb{F}, x, y \in \mathbb{R}^+, x \leq y$; this ensures that our network respects the FIFO property when the functions are interpreted as travel times (see Section 1.2.1). The juxtaposition $f \oplus g$ of two functions $f, g \in \mathbb{F}$ is a function $\in \mathbb{F}$ defined as $(f \oplus g)(\tau) = f(\tau) + g(f(\tau) + \tau) \; \forall \tau \in \mathbb{R}^+$. Note that this operation is neither commutative nor associative, and should be evaluated from left to right; that is, $f \oplus g \oplus h = (f \oplus g) \oplus h$. The minimum $\min\{f, g\}$ of two functions $f, g \in \mathbb{F}$ is a function $\in \mathbb{F}$ such that $(\min\{f, g\})(\tau) = \min\{f(\tau), g(\tau)\} \; \forall \tau \in \mathcal{T}$. We define the lower bound of f as $\underline{f} = \min_{\tau \in \mathcal{T}} f(\tau)$, and the upper bound as $\overline{f} = \max_{\tau \in \mathcal{T}} f(\tau)$.

Consider a directed graph $G = (V, A)$, where the cost of an arc (u, v) is a time-dependent function given by a function $c : A \to \mathbb{F}$; for simplicity, we will write $c(u, v, \tau)$ instead of $c(u, v)(\tau)$ to denote the cost of the arc (u, v) at time $\tau \in \mathcal{T}$. We define $\lambda, \rho : A \to \mathbb{R}^+$ as $\lambda = \underline{c}$ and $\rho = \overline{c}$, i.e. $\forall (u, v) \in A \; \lambda(u, v) = \underline{c}(u, v)$ and $\rho(u, v) = \overline{c}(u, v)$; we assign their own symbol to these two functions because they will be used very often in the following.

We denote the distance between two nodes $s, t \in V$ with departure from s at time $\tau_0 \in \mathcal{T}$ as $d(s, t, \tau)$. The distance function between s and t is defined as $d_*(s, t) : \mathcal{T} \to \mathbb{R}^+, d_*(s, t)(\tau) = d(s, t, \tau)$. We denote by G_λ the graph G weighted by the lower bounding function λ; the distance between two nodes s, t on G_λ is denoted by $d_\lambda(s, t)$. Similarly, we denote the graph G weighted by ρ as G_ρ.

Given a path $p = (s = v_1, \ldots, v_i, \ldots, v_j, \ldots, v_k = t)$, its *time-dependent cost* is defined as $\gamma(p) = c(v_1, v_2) \oplus c(v_2, v_3) \oplus \cdots \oplus c(v_{k-1}, v_k)$. Its time-dependent cost with departure time at $\tau_0 \in \mathcal{T}$ is denoted as $\gamma(p, \tau_0) = \gamma(p)(\tau_0)$. We denote the subpath of p from v_i to v_j by $p_{|v_i \to v_j}$. The concatenation of two paths p and q is denoted by $p + q$.

We call $\overline{G}$ the reverse graph of G, i.e. $\overline{G} = (V, \overline{A})$ where $\overline{A} = \{(u, v) | (v, u) \in A\}$. For $V' \subset V$, we define $A[V'] = \{(u, v) \in A | u \in V', v \in V'\}$ as the set of arcs with both endpoints in V'. Correspondingly, the subgraph of G induced by V' is $G[V'] = (V', A[V'])$. We define the *union* between two graphs $G_1 = (V_1, A_1)$ and $G_2 = (V_2, A_2)$ as $G_1 \cup G_2 = (V_1 \cup V_2, A_1 \cup A_2)$.

We can now formally state the Time-Dependent Shortest Path Problem:

> TIME-DEPENDENT SHORTEST PATH PROBLEM(TDSPP): given a directed graph $G = (V, A)$ with cost function $c : A \to \mathbb{F}$ as defined above, a source node $s \in V$, a destination node $t \in V$ and a departure time

$\tau_0 \in \mathcal{T}$, find a path $p = (s = v_1, \ldots, v_k = t)$ in G such that its *time-dependent cost* $\gamma(p, \tau_0)$ is minimum.

We will assume that our problem is to find the fastest path between two nodes with departure at a given time; the "backward" version of this problem, i.e. finding the fastest path between two nodes with *arrival* at a given time, can be solved with the same method (see (39)).

1.2.1 The FIFO property

The First-In-First-Out property states that for each pair of time instants $\tau, \tau' \in \mathcal{T}$ with $\tau' > \tau$:

$$\forall\ (u,v) \in A\ c(u,v,\tau) + \tau \leq c(u,v,\tau') + \tau',$$

The FIFO property is also called the *non-overtaking property*, because it basically says that if T_1 leaves u at time τ and T_2 at time $\tau' > \tau$, T_2 cannot arrive at v before T_1 using the arc (u,v). Note that our choice of the function space for the time-dependent arc cost function in Section 1.2 ensures that the FIFO property holds. Although FIFO networks are useful for the study of those means of transportation where overtaking is rare (such as trains), modelling of car transportation yields networks which do not necessarily have the FIFO property. For the TDSPP, the FIFO assumption is usually necessary in order to mantain an acceptable level of complexity: the SPP in time-dependent FIFO networks is polynomially solvable (85), even in the presence of traffic lights (5), while it is NP-hard in non-FIFO networks (117).

In Part I we will deal with time-dependent graphs for which the FIFO property holds. This is motivated by the fact that the real-world time-dependent data provided by the Mediamobile company[1] consists in functions which satisfy the FIFO property. However, at the beginning of this thesis this was not clear because data gathering and manipulation was still in progress. Hence, the parts of this work dealing with mathematical programming are motivated by the study of non-FIFO networks. The original idea was to consider only FIFO functions, so that the TDSPP is polynomially solvable, and to use a mathematical programming formulation of the TDSPP on non-FIFO networks in order to verify the quality of the solutions found.

1.2.2 Choice of the cost functions

In order to implement an efficient algorithm for shortest paths computations on time-dependent graphs we must be able to efficiently carry out several operations between time-dependent functions, e.g.: computing the composition

[1] `http://www.v-trafic.com`

and the minimum of two functions, obtaining lower and upper bounds (see Section 1.4.3). Moreover, the functions should be as quick as possible to evaluate. The practical difficulty of dealing with time-dependent cost function depends on the complexity of the cost function (40). For real-time applications, we want to keep this difficulty as low as possible; thus, a natural choice is to use piecewise linear functions to model arc costs, which allow for some flexiblity while being simple to treat algorithmically. Furthermore, piecewise linear functions have the advantage that the FIFO property can be easily enforced: it is straightforward to note that the condition $f(x) + x \leq f(y) + y \ \forall x \leq y$ translates to $\frac{df(x)}{dx} \geq -1$.

Although all theoretical considerations are valid for general cost functions, through the rest of this work when dealing with FIFO networks we will assume that from a practical point of view the time-dependent cost functions on arcs can be represented by piecewise linear functions. In particular, this holds through Part I.

When lifting the restrictions on the arc costs, a reasonable model to take into account perturbations due to traffic on an arc is to consider a constant cost, which represents the travelling time in traffic-free conditions, plus a summation of Gaussian functions, each one centered on a traffic congestion. Formally, for each arc $(i, j) \in A$ we have:

$$c(i, j, \tau) = c_{ij} + \sum_{k=1}^{h} a_k e^{-\frac{(\tau-\mu_k)^2}{2\sigma_k^2}},$$

where c_{ij} is the travelling time over arc (i, j) in uncongested hours, and h is the number of traffic congestions over one day. Each congestion is centered at time μ_k, and has in practice no effect more than $3\sigma_k$ away from the mean μ_k. Note that this cost function is not necessarily FIFO, thus we cannot employ Dijkstra-like algorithms. Therefore, we will deal with it through a mathematical programming formulation.

1.3 Mathematical Programming Formulations for the TDSPP

The TDSPP in FIFO networks can be efficiently solved in a combinatorial way, as we will see in Part I. However, it can also be modeled as a mathematical program, and solved with general-purpose methods for mathematical programs. This will be the subject of Part II. In this section we define a mathematical program that models the TDSPP with arbitrary cost functions.

The rest of this section is organized as follows. In Section 1.3.1 we give a definition of mathematical program taken from the literature, identifying different classes of mathematical programs. In Section 1.3.2 we give a mathematical

programming formulation for the TDSPP with arbitrary cost functions. In Section 1.3.3 the size of the proposed formulations is analyzed, and a reasonable model for the cost functions is proposed.

1.3.1 Definition of mathematical program

Wikipedia[2] defines mathematical programming as:

> [...] the study of problems in which one seeks to minimize or maximize a real function by systematically choosing the values of real or integer variables from within an allowed set. This (a scalar real valued objective function) is actually a small subset of this field which comprises a large area of applied mathematics and generalizes to study of means to obtain "best available" values of some objective function given a defined domain where the elaboration is on the types of functions and the conditions and nature of the objects in the problem domain.

Typically, mathematical programs are cast in the form:

$$\left.\begin{array}{rr} \min & f(x) \\ \text{subject to:} & \\ \forall j \in M & g_j(x) \leq 0 \\ & x^L \leq x \leq x^U \\ & x \in X \end{array}\right\} (P)$$

where X is a cartesian product of continuous and discrete intervals. In this case, we have a single objective function f, a set M of constraints g_j, a vector of variable lower and upper bounds x^L, x^U, not necessarily finite. The meaning of the mathematical program (P) is that we seek, among all points $x \in X$ which satisfiy the constraints $g_j(x) \leq 0 \ \forall j \in M$, $x^L \leq x \leq x^U$, the one that yields the smallest value of $f(x)$.

A formal definition of mathematical program is given in (93; 94). The definition is such that it easily translates into a data structure that can be implemented on a computer. Let $\mathbb{P}$ be the set of all mathematical programs, and $\mathbb{M}$ be the set of all matrices. We recall that, given a directed graph $G = (V, A)$ and a node $v \in V$, $\delta^+(v)$ indicates the set of vertices u such that $(v, u) \in A$, and $\delta^-(v)$ denotes the set of vertices u such that $(u, v) \in A$. The definition of a mathematical program given in (93; 94) is as follows.

Definition 1.3.1. *Given an alphabet $\mathcal{L}$ consisting of countably many alphanumeric names $N_{\mathcal{L}}$ and operator symbols $O_{\mathcal{L}}$, a mathematical programming formulation P is a 7-tuple $(\mathcal{P}, \mathcal{V}, \mathcal{E}, \mathcal{O}, \mathcal{C}, \mathcal{B}, \mathcal{T})$, where:*

[2] `http://www.wikipedia.org`

- *$\mathcal{P} \subset N_{\mathcal{L}}$ is the sequence of parameter symbols: each element $p \in \mathcal{P}$ is a parameter name;*
- *$\mathcal{V} \subset N_{\mathcal{L}}$ is the sequence of variable symbols: each element $v \in \mathcal{V}$ is a variable name;*
- *$\mathcal{E}$ is the set of expressions: each element $e \in \mathcal{E}$ is a DAG $e = (V_e, A_e)$ such that:*
  - *(a) $V_e \subset \mathcal{L}$ is a finite set*
  - *(b) there is a unique vertex $r_e \in V_e$ such that $\delta^-(r_e) = \emptyset$ (such a vertex is called the root vertex)*
  - *(c) vertices $v \in V_e$ such that $\delta^+(v) = \emptyset$ are called leaf vertices and their set is denoted by $\mathcal{L}(e)$; all leaf vertices are such that $v \in \mathcal{P} \cup \mathcal{V} \cup \mathbb{R} \cup \mathbb{P} \cup \mathbb{M}$*
  - *(d) $\forall v \in V_e : \delta^+(v) \neq \emptyset \Rightarrow v \in O_{\mathcal{L}}$*
  - *(e) two weight functions $\chi, \zeta : V_e \to \mathbb{R}$ are defined on V_e: $\chi(v)$ is the node coefficient and $\zeta(v)$ is the node exponent of the node v; for any vertex $v \in V_e$, we let $\tau(v)$ be the symbolic term of v: namely, $v = \chi(v)\tau(v)^{\zeta(v)}$.*

  Elements of $\mathcal{E}$ are sometimes called expression trees; nodes $v \in O_{\mathcal{L}}$ represent an operation on the nodes in $\delta^+(v)$, denoted by $v(\delta^+(v))$, with output in $\mathbb{R}$;
- *$\mathcal{O} \subset \{-1, 1\} \times \mathcal{E}$ is the sequence of objective functions; each objective function $o \in \mathcal{O}$ has the form (d_o, f_o) where $d_o \in \{-1, 1\}$ is the optimization direction (-1 stands for minimization, $+1$ for maximization) and $f_o \in \mathcal{E}$;*
- *$\mathcal{C} \subset \mathcal{E} \times \mathcal{S} \times \mathbb{R}$ (where $\mathcal{S} = \{-1, 0, 1\}$) is the sequence of constraints c of the form (e_c, s_c, b_c) with $e_c \in \mathcal{E}, s_c \in \mathcal{S}, b_c \in \mathbb{R}$:*

$$c \equiv \begin{cases} e_c \leq b_c \text{ if } s_c = -1 \\ e_c = b_c \text{ if } s_c = 0 \\ e_c \geq b_c \text{ if } s_c = 1; \end{cases}$$

- *$\mathcal{B} \subset \mathbb{R}^{|\mathcal{V}|} \times \mathbb{R}^{|\mathcal{V}|}$ is the sequence of variable bounds: for all $v \in \mathcal{V}$ let $\mathcal{B}(v) = [L_v, U_v]$ with $L_v, U_v \in \mathbb{R}$;*
- *$\mathcal{T} \subset \{0, 1, 2\}^{|\mathcal{V}|}$ is the sequence of variable types: for all $v \in \mathcal{V}$, v is called a continuous variable if $\mathcal{T}(v) = 0$, an integer variable if $\mathcal{T}(v) = 1$ and a binary variable if $\mathcal{T}(v) = 2$.*

Given an expression tree DAG $e = (V_e, A_e)$ with root node $r(e)$ and whose leaf nodes are elements of $\mathbb{R}$ or of $\mathbb{M}$, the evaluation of e is the numerical output of the operation represented by the operator node in node r applied to all nodes adjacent to r. For leaf nodes belonging to $\mathbb{P}$, the evaluation is not defined; the

mathematical program in the leaf node must first be solved and a relevant optimal value must replace the leaf. An algorithm to evaluate expression trees is given in (94).

Definition 1.3.1 states that a mathematical program consists in a set of variables with an associated type and lower/upper bounds, a set of parameters, a set of equality/inequality constraints, and a set of objective functions, each one with an associated optimization direction (minimization/maximization).

Based on Definition 1.3.1, we distinguish several classes of mathematical programs. In this thesis, we are interested in the following categories.

- Linear Programs: a mathematical programming problem P is a Linear Program (LP) if $|\mathcal{O}| = 1$, e is a linear form for all $e \in \mathcal{E}$, and $\mathcal{T}(v) = 0$ for all $v \in V$. In other words, a LP has only one objective value, linear objective function and constraints, and all variables are continuous.

- Mixed-Integer Linear Programs: a mathematical programming problem P is a Mixed-Integer Linear Program (MILP) if $|\mathcal{O}| = 1$ and e is a linear form for all $e \in \mathcal{E}$. In other words, a MILP has only one objective value, linear objective function and constraints, and variables can be both continuous and discrete.

- Nonlinear Programs: a mathematical programming problem P is a Nonlinear Program (NLP) if $|\mathcal{O}| = 1$ and $\mathcal{T}(v) = 0$ for all $v \in V$. In other words, a NLP has only one objective value and all variables are continuous, while the objective function and the constraints can be arbitrary linear/nonlinear expressions.

- Mixed-Integer Nonlinear Programs: a mathematical programming problem P is a Mixed-Integer Nonlinear Program (MINLP) if $|\mathcal{O}| = 1$. In other words, a MINLP has only one objective value; variables can be both continuous and discrete, while the objective function and the constraints can be arbitrary linear/nonlinear expressions.

Within the class of NLPs (respectively, MINLPs), we distinguish between convex NLPs (MINLPs) if e represents a convex function for all $e \in \mathcal{E}$, whereas it is a nonconvex NLP (MINLP) otherwise. In general, solving LPs and convex NLPs is considered easy, and solving MILPs, nonconvex NLPs and convex MINLPs (cMINLPs) is considered difficult. Solving nonconvex MINLPs involves difficulties arising from both nonconvexity and integrality, and it is considered the hardest problem of all.

A mathematical program may also have multiple objective functions, which adds to the complexity of the problem. In fact, in a mathematical sense it is not clear how a solution can be optimal with respect to more than one objective function, if these are conflicting. In this case, one is often interested in the set of non-dominated solutions, i.e. the set of Pareto optima. Intuitively, this

consists in the set of solutions such that each one is better than the other ones for at least one of the considered optimization criteria. However, in this work we are mainly interested in single objective optimization; we refer the reader to (55) for an introduction to multi-objective optimization.

1.3.2 Formulation of the TDSPP

We seek to derive mathematical programming formulations for the TDSPP under different assumptions. It is natural to start with a formulation for the shortest paths problem on static graphs, and then add time-dependency into the model. A classical formulation for the SPP (see (87)) is the following. Let $M \in \{-1, 0, 1\}^{|V|\times|A|}$ be the incidence matrix of G, i.e., a matrix whose element m_{ij}^v is: +1 if $v = i$, i.e. if arc (i, j) is in the forward star of node v, -1 if $v = j$, and 0 otherwise. Suppose c_{ij} is the cost of arc $(i, j) \in A$. We consider a network flow problem (4) with demands $b_v = 1$ for $v = s$, $b_v = -1$ for $v = t$ and $b_v = 0\ \forall v \in V \setminus \{s, t\}$:

$$\left.\begin{array}{rlcl} \min & \sum_{(i,j)\in A} c_{ij}x_{ij} & & \\ \forall v \in V & \sum_{(i,j)\in A} m_{ij}^v x_{ij} & = & b_v \\ \forall (i,j) \in A & x_{ij} & \in & \{0,1\} \end{array}\right\} (SPP)$$

This is equivalent to introducing one unit of flow at the source node, and requiring that this unit reaches the destination while passing through arcs that minimize the total cost. (SPP) is a linear program with both integer and continuous variables. It is well known that, since the constraint matrix of (SPP) is unimodular, then all solutions to the linear relaxation of (SPP) are integral, assuming that the costs c_{ij} are integral. As a consequence, (SPP) is an easy problem. We can extend the above formulation in order to model the TDSPP, by introducing extra variables $\tau_v \forall v \in V$ which represent the arrival time at node v.

$$\left.\begin{array}{rlcl} \min & \tau_t & & \\ \forall v \in V & \sum_{(i,j)\in A} m_{ij}^v x_{ij} & = & b_v \\ \forall (i,j) \in A & x_{ij}(\tau_i + c(i,j,\tau_i)) & \leq & \tau_j \\ \forall (i,j) \in A & x_{ij} & \in & \{0,1\} \\ \forall v \in V & \tau_i & \geq & 0 \end{array}\right\} (TDSPP)$$

In the above formulation, $c(i, j, \tau_i)$ represents the cost of arc (i, j) at time τ_i, following the notation introduced in Section 1.2. $(TDSPP)$ contains the flow conservation constraints of a network flow problem, but has additional constraints that link the arrival time at node j with the departure time from node i, if the arc (i, j) is chosen. It is immediate to notice that the constraint matrix is no longer unimodular. By the FIFO property, and since we are minimizing the arrival time τ_t at node t, for all arcs which are in the shortest path (i.e. $x_{ij} = 1$) the corresponding arrival time definition constraints are satisfied at equality, which implies $\tau_i + c(i, j, \tau_i) = \tau_j$. This proves correctness.

The difficulty of solving $(TDSPP)$ depends on the form of $c(i,j,\tau_i)$; we will discuss this issue later in Section 1.3.3. $(TDSPP)$ assumes that there is no waiting at nodes, which is a necessary condition for optimal solutions to the TDSPP in FIFO networks. The model can be amended so as to yield the optimal solution even in the non-FIFO case; we introduce variables d_v to indicate the departure time from node v. Note that, if the FIFO property is satisfied, we have $d_v = \tau_v$, but equality does not necessarily hold in the general (non-FIFO) scenario. The problem becomes:

$$\left.\begin{array}{rrcl} \min & \tau_t & & \\ \forall v \in V & \sum_{(i,j)\in A} m_{ij}^v x_{ij} & = & b_v \\ \forall v \in V & \tau_v & \leq & d_v \\ \forall (i,j) \in A & x_{ij}(d_i + c(i,j,d_i)) & \leq & \tau_j \\ \forall (i,j) \in A & x_{ij} & \in & \{0,1\} \\ \forall v \in V & \tau_i & \geq & 0 \end{array}\right\} (GTDSPP)$$

The complicating constraint in $(GTDSPP)$, as well as in $(TDSPP)$, is the definition of the arrival time at node j if arc (i,j) is in the shortest path: $\forall (i,j) \in A\ x_{ij}(d_i + c(i,j,d_i)) \leq \tau_j$. These constraints involve a product between the binary variable x_{ij} and the continuous variable d_i, which can be reformulated in linear form by introducing extra variables and constraints (94), and the product between x_{ij} and $c(i,j,d_i)$, whose difficulty depends on the form of $c(i,j,d_i)$. Obviously, we would like to keep $(GTDSPP)$ as easy as possible. If $c(i,j,d_i)$ is a piecewise linear function as assumed in Section 1.2.2, then $c(i,j,d_i)$ can be written in linear form by introducing extra binary variables, one for each breakpoint. These binary variables serve the purpose of selecting which piece of the piecewise linear function is active at the given point in which we want to calculate the value of the function. Therefore, the constraints $\forall (i,j) \in A\ x_{ij}(d_i + c(i,j,d_i)) \leq \tau_j$ are linear constraints which involve products between binary variables and continuous or binary variables. Following (94), all these products can be expressed in linear form by adding some variables and defining constraints. Thus, $(GTDSPP)$ is a MILP. However, it may also be interesting to consider nonlinear cost functions $c(i,j,d_i)$ (see Section 1.3.3). In this case, $(GTDSPP)$ is a (possibly nonconvex) MINLP.

The greatest advantage of a mathematical programming formulation for the TDSPP is its flexibility. Not only we are able to consider arbitrary cost functions, but we can also take into account additional complicating constraints which would be very difficult to deal with when using Dijkstra-like algorithms. One such example is prohibited turnings on the shortest path. Together with the network G, we are also given a list of arc pairs (prohibited turnings) such that the head of the first arc is the tail of the second; e.g. $((u,v),(v,w))$. In this case, we want to compute the shortest path between two nodes s and t such that the path does not contain two consecutive arcs that represent a prohibited turning. In road network applications, this is useful to model road junctions where,

for instance, a left-turn is forbidden. Dijkstra's algorithm cannot be applied in a straightforward manner if we want to take into account prohibited turnings. The problem has been tackled in (14; 15) by computing label-constrained shortest paths; however, this approach requires a significant amount of additional computations and slows down Dijkstra's algorithm. Moreover, it implies the definition of a regular language that does not accept prohibited turnings and where each node of the graph is associated with a symbol of the alphabet. On the other hand, prohibited turnings can be modeled in a very simple way with the mathematical formulation $(GTDSPP)$: it suffices to add the constraint $x_{uv} + x_{vw} \leq 1$ for each prohibited turning $((u,v),(v,w))$.

1.3.3 Analysis of the formulations

The size of the formulation $(GTDSPP)$ depends on several factors. Clearly, the size of the graph G plays a most important role, because some variables and constraints are defined for each node and arc that appear in the network. In the case of piecewise linear cost functions, the total number of breakpoints in the network is also important, as extra variables and constraints have to be added for each one of these breakpoints. If we assume that $c(i,j,\tau_i)$ is nonlinear, as is the case if we use the summation of Gaussians model proposed in Section 1.2.2, then solution algorithms for MINLPs (23; 131; 132) typically add extra variables, depending on the expression of the cost function c. It is likely that, for a road network with millions of nodes and arcs, the resulting formulation $(GTDSPP)$ would have several millions or billions of variables and constraints. Therefore, there is no hope of solving it to optimality with existing exact algorithms within the short time slots allowed by real-time applications. However, this formulation may be useful for practical purposes as a mean to study networks which are difficult to treat with Dijkstra's algorithm, such as non-FIFO networks or networks with general nonlinear time-dependent costs, possibly restricting the size of the analyzed graph. This may allow to underline the differences between FIFO and non-FIFO scenarios, as well as understanding which models are more meaningful from a user point of view. We recall that at the beginning of this thesis it was not clear whether the real-world time-dependent data would satisfy the FIFO property or not, and it was not clear if it would be highly nonlinear or it could be modeled with piecewise linear functions. This is because data gathering and manipulation was still in progress. Therefore, we wanted to have the possibility of studying general non-FIFO networks as a mean to verify the quality of the solutions found by simplifications of the problem. To do so, we investigated efficient algorithms to quickly find good (hopefully, optimal) solutions to both MILPs and MINLPs.

1.4 Related Work

The Shortest Path Problem (SPP) is one of the best studied combinatorial optimization problems in the literature (4; 128). Many ideas have been proposed for the computation of point-to-point shortest paths on static graphs (see (133; 127) for a review), and there are algorithms capable of finding the solution in a matter of a few microseconds (16); adaptations of those ideas for dynamic scenarios, i.e. where arc costs are updated at regular intervals, have been tested as well (46; 126; 134; 114). The time-dependent variant of the SPP has received much less attention throughout the years. In this section we survey some of the results in this field, as well as a few works on speedup techniques for the SPP on static graphs which will be frequently referred to in the following. As some of the ideas we will describe deal with static graphs (i.e. not time-dependent), throughout this section we will denote by $c(u,v)$ the cost of an arc $(u,v) \in A$ in the static case, and by $d(u,v)$ the length of the shortest path between u and v in the same scenario.

The rest of this section is organized as follows. In Section 1.4.1 we discuss some pre-1980 studies on the TDSPP and seminal work on reoptimization techniques for graphs with dynamic arc weights, which lead to insight on future developments of the TDSPP. In Section 1.4.2 we describe Dijkstra's algorithm, which laid the foundations for all following shortest paths algorithms. In Section 1.4.3 we report the main ideas of a label-correcting algorithm which computes a cost function that gives the distance between two nodes for each time instant on a time-dependent graph. In Section 1.4.4 we analyse some of the most important hierarchical speedup techniques for the point-to-point SPP on static graphs. In Section 1.4.5 we discuss the A^* algorithm for goal-directed search, and an A^*-based efficient algorithm for shortest paths computations on road networks which is the basic ingredient for our main algorithm (see Chapter 3). In Section 1.4.6 we review the recently developed SHARC algorithm, which currently represents the state-of-the-art of unidirectional shortest paths algorithms on time-dependent graphs.

1.4.1 Early history

One of the main direct application of shortest path type problems is in transportation theory. A lot of early work (1950 – 1960) was carried out on related topics at the RAND corporation, but it was mostly to do with transportation network analysis (on dynamic networks where the capacities changed according to traffic congestion) rather than the shortest path to be chosen by any individual driver (21).

The first citation we could find concerning the TDSPP is (36) (a good review of this paper can be found in (53), p. 407): a recursive formula is given to establish the minimum time to travel to a given target starting from a given source at

time τ. It is shown that if travel times take on integer positive values then the procedure terminates with the shortest path from all nodes to a given destination. Using the notation introduced in Section 1.2, let $t \in V$ be the destination node, and s the starting node. The procedure is based on the formula

$$\begin{aligned} d(s,t,\tau) &= \min_{v \in V:(s,v)\in A} \{c(s,v,\tau) + d(v,t,\tau + c(s,v,\tau))\} \\ d(t,t,\tau) &= 0. \end{aligned}$$

In (53), Dijkstra's algorithm (49) (see Section 1.4.2) is extended to the dynamic case, but the FIFO property (Section 1.2.1), which is necessary to prove that Dijkstra's algorithm terminates with a correct shortest paths tree on time-dependent networks, is not mentioned.

Early studies on general transportation networks were mostly motivated by transportation planning, i.e. network analysis in order to optimize investments to improve the current road network; see (75) for a survey. This required to study the effect of modifying a link on the routes chosen by the network users. A road network was modeled as a graph where each link had an associated travelling time and a capacity, and nodes corresponded to entry points on the road network of particular zones (75). Thus, only *interzonal* travelling times affected the road network. The number of individuals that chose a particular source-destination pair at each time of the day was supposed to be known by demographical studies or trip generation techniques, and routes were assigned computing the shortest paths tree rooted at each node of the network. The first case to be analysed is the shortening of a link (102; 107), i.e. the decrease of its associated travelling time: it is observed that in this situation the length of the shortest path between two nodes s,t will decrease only if the shortest path between s,t passing through the affected arc is shorter than the previous solution. Thus, if (u,v) is the link to be shortened, $d(s,t)$ is the initial cost of the shortest path between two nodes s,t, and $c'(u,v)$ is the new cost of arc (u,v), the new shortest path distances can be computed as $d'(s,t) = \min\{d(s,t), d(s,u)+c'(u,v)+d(v,t)\}$. The *method of competing links* (74) analysed the effect of an arbitrary change in the cost of a link in a cutset: the graph was partitioned in two sets Z_1, Z_2, and if we call C the set of arcs connecting the two node sets then the travelling time between two nodes $s \in Z_1, t \in Z_2$ was computed as

$$\min_{(p,q)\in C}(d(s,p) + d(p,q) + d(q,t)),$$

where again $d(i,j)$ is the cost of the shortest path from i to j. As only the costs of arcs in the cutset C were allowed to change, the new shortest paths trees were easily computed.

The first attempts to solving the SPP on dynamic graphs (i.e. arc costs are allowed to change) relied on reoptimization techniques: in particular, (108) considers the problem of finding the shortest path cost matrix when only one arc

of the input graph changes its cost. The same problem was investigated a few years later in (50). (63) addresses the SPP on dynamic graphs where either an arc changes its cost or a different root node is selected, and lays the foundation for future work; it proposes a procedure to reduce the complexity of Dial's implementation (48) of Dijkstra's algorithm. The number of comparisons needed by Dial's implementation depends on the cost of the longest shortest path from the root to all other nodes of the graph; in order to reduce this cost, (63) modifies the length of all arcs with the formula $c'(i,j) = c(i,j) + \pi_i - \pi_j$, where c' is the new cost function, c is the old cost function, and $\pi_i \ \forall i \in V$ is a positive integer such that $c'(i,j) \geq 0 \ \forall (i,j) \in A$. It is noted that a transformation of this kind does not modify which arcs appear on a shortest path, and was first proposed in (115) in order to get non-negative arc costs on graphs with $c(i,j) < 0$ for some $(i,j) \in A$. This observation is of fundamental importance for the A^* algorithm (see Section 1.4.5). The interpretation of the vector $(\pi_1, \ldots, \pi_{|V|})$ as a dual feasible solution to the SPP is due to (20).

1.4.2 Dijkstra's algorithm

Dijkstra's algorithm (49) solves the single source SPP in static directed graphs with non-negative weights in polynomial time. The algorithm can easily be generalized to the time-dependent case (53). Dijkstra's algorithm is a so-called labeling method.

The *labeling method* for the SPP (60) finds shortest paths from the source to all vertices in the graph; the method works as follows: for every vertex v it maintains its distance label $\ell[v]$, parent node $p[v]$, and status $S[v]$ which may be one of the following: `unreached`, `explored`, `settled` . Initially $\ell[v] = \infty$, $p[v] = NIL$, and $S[v] =$ `unreached` for every vertex v. The method starts by setting $\ell[s] = 0$ and $S[s] =$ `explored`; while there are labeled (i.e. explored) vertices, the method picks an `explored` vertex v, relaxes all outgoing arcs of v, and sets $S[v] =$ `settled`. To relax an arc (v,w), one checks if $\ell[w] > \ell[v] + c(v,w)$ and, if true, sets $\ell[w] = \ell[v] + c(v,w)$, $p(w) = v$, and $S(w) =$ `explored`. If the graph does not contain cycles with negative cost, the labeling method terminates with correct shortest path distances and a shortest path tree. The algorithm can be extended to the time-dependent case on FIFO networks by a simple modification of the arc relaxation procedure: if τ_0 is the departure time from the source node, we check if $\ell[w] > \ell[v] + c(v,w,\tau_0 + \ell[v])$ and, if true, set $\ell[w] = \ell[v] + c(v,w,\tau_0 + \ell[v])$, $p[w] = v$, and $S[w] =$ `explored`. The efficiency of the label-setting method depends on the rule to choose a vertex to scan next. We say that $\ell[v]$ is exact if it is equal to the distance from s to v; it is easy to see that if one always selects a vertex v such that, at the selection time, $\ell[v]$ is exact, then each vertex is scanned at most once. In this case we only need to relax arcs (v,w) where w is not `settled`, and the algorithm is called *label-setting*. Dijkstra (49) observed that if the cost function c is non-negative and v is an explored vertex with the smallest distance

label, then $\ell[v]$ is exact; so, we refer to the labeling method with the minimum label selection rule as Dijkstra's algorithm. If c is non-negative then Dijkstra's algorithm scans vertices in nondecreasing order of distance from s and scans each vertex at most once; for the point-to-point SPP, we can terminate the labeling method as soon as the target node is `settled`. The algorithm requires $O(|A| + |V| \log |V|)$ amortized time if the queue is implemented as a Fibonacci heap (62); with a binary heap, the running time is $O((|E| + |V|) \log |V|)$.

One basic variant of Dijkstra's algorithm for the point-to-point SPP is bidirectional search; instead of building only one shortest path tree rooted at the source node s, we also build a shortest path tree rooted at the target node t on the reverse graph $\overline{G}$. As soon as one node v becomes `settled` in both searches, we are guaranteed that the concatenation of the shortest $s \rightarrow v$ path found in the forward search and of the shortest $v \rightarrow t$ path found in the backward search is a shortest $s \rightarrow t$ path. Since we can think of Dijkstra's algorithm as exploring nodes in circles centered at s with increasing radius until t is reached (see Figure 1.1), the bidirectional variant is faster because it explores nodes in two circles centered at both s and t, until the two circles meet (see Figure 1.2); the area within the two circles, which represents the number of explored nodes, will then be smaller than in the unidirectional case, up to a factor of two.

Dijkstra's algorithm applied to time-dependent FIFO networks has been optimized in various ways (29; 31). We note here that in the time-dependent scenario bidirectional search cannot be applied, since the arrival time at destination node is unknown. We also remark that all speedup techniques based on finding shortest paths in Euclidean graphs (130) cannot be applied either, since the typical arc cost function, the arc travelling time at a certain time of the day, does not yield a Euclidean graph.

1.4.3 Label-correcting algorithm

On a time-dependent graph, we can use a *label-correcting* algorithm to compute $d_*(s,t)$ (Section 1.2) instead of $d(s,t,\tau)$ for $\tau \in \mathcal{T}$; label-correcting implies that the label of a node is not fixed even after the node is extracted from the priority queue, in that a node may be reinserted multiple times, unlike Dijkstra's algorithm. We refer to (40) for an excellent starting point on the efficient implementation of TDSPP algorithms. We describe here a label-correcting algorithm (40) to compute the cost function associated with the shortest path between two nodes $s, t \in V$. Such an algorithm can be implemented similarly to Dijkstra's algorithm, but using arc *cost functions* instead of arc *lengths*. The label $\ell(v)$ of a node v is a scalar for plain Dijkstra's algorithm, whereas in this case each label is a function of time. In particular, at termination we want $\ell(v) = d_*(s,v)$. We initialize the algorithm assigning constant functions as labels: $\forall \tau \in \mathcal{T}\ \ell(s)(\tau) = 0$ and $\ell(v)(\tau) = \infty\ \forall v \in V$. At each iteration we extract the node u with minimum $\underline{\ell}(u)$ from the priority queue, and relax adjacent

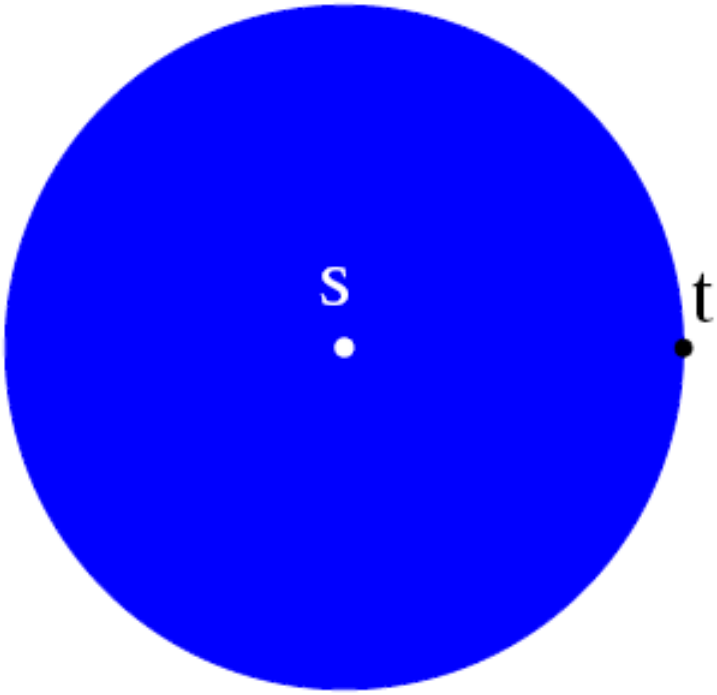


Figure 1.1: Schematic representation of Dijkstra's algorithm search space

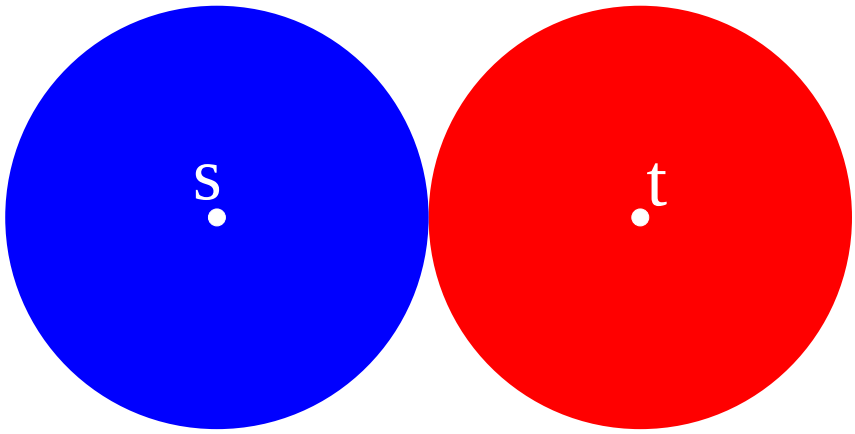


Figure 1.2: Schematic representation of bidirectional Dijkstra's algorithm search space.

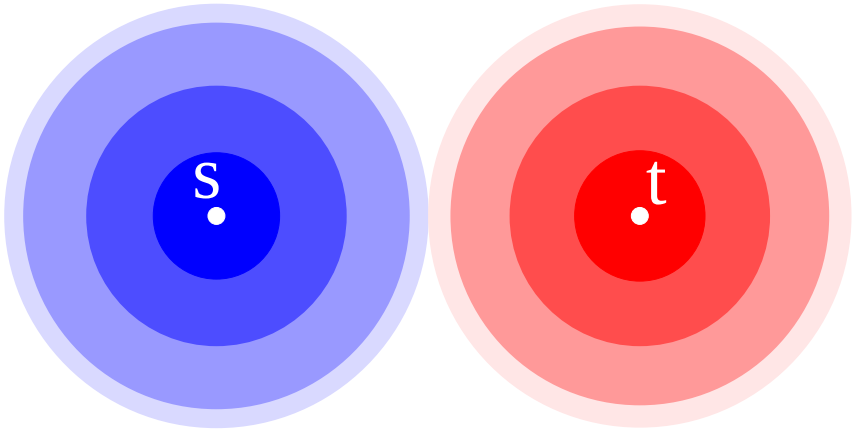


Figure 1.3: Schematic representation of a hierarchical speedup technique search space

edges: for each $(u,v) \in A$, a temporary label $t(v) = \ell(u) \oplus c(u,v)$ is created. Then if $t(v)(\tau) \geq \ell(v)(\tau)$ for all $\tau \in \mathcal{T}$ does not hold, the arc (u,v) yields an improvement for at least one time instant. Hence, we update $\ell(v) = \min\{\ell(v), t(v)\}$. The algorithm can be stopped as soon as we extract a node u such that $\underline{\ell}(u) \geq \overline{\ell}(t)$. An interesting observation from (40) is that the running time of this algorithm depends on the complexity of the cost functions associated with arcs.

1.4.4 Hierarchical speedup techniques for static road networks

Many hierarchical speedup techniques have been developed for the SPP on static graphs. The main idea is to preconstruct a graph hierarchy where each level is smaller then the previous one, i.e. it has fewer nodes; shortest paths queries start at the bottom level and are then carried out exploring the hierarchy levels in ascending order, so that most of the search is carried out on the topmost level. Since the number of nodes at each level shrinks rapidly as we progress upwards into the hierarchy, the total number of explored nodes is considerably smaller than in a plain appplication of Dijkstra's algorithm (see Figure 1.3). Due to the inherent bidirectional nature of these algorithms, these approaches only work on static graphs.

1.4.4.1 Highway Hierarchies

The Highway Hierarchies algorithm (HH) (129; 124; 125) is a fast, hierarchy-based shortest paths algorithm which works on static directed graphs. HH is specially suited to efficiently finding shortest paths in large-scale networks, and

has been the first algorithm to report average query times of a few milliseconds on continental sized road networks.

A set of shortest paths is *canonical* if, for any shortest path p in the set, $p = (u_1, \ldots, u_i, \ldots, u_j, \ldots, u_k)$, the canonical shortest path between u_i and u_j is a subpath of p. Dijkstra's algorithm can easily be modified to output a canonical shortest paths tree (129).

The HH algorithm works in two stages: a time-consuming pre-processing stage to be carried out only once, and a fast query stage to be executed at each shortest path request. Let $G^0 = G$. During the first stage, a highway hierarchy is constructed, where each hierarchy level G^l, for $1 \leq l \leq L$, is a modified subgraph of the previous level graph G^{l-1} such that no canonical shortest path in G^{l-1} lies entirely outside the current level for all sufficiently distant path endpoints: this ensures that all queries between far endpoints on level $l-1$ are mostly carried out on level l, which is smaller, thus speeding up the search. Each shortest path query is executed by a multi-level bidirectional Dijkstra algorithm: two searches are started from the source and from the destination, and the query is completed shortly after the search scopes have met; at no time do the search scopes decrease hierarchical level. Intuitively, path optimality is due to the fact that by hierarchy construction there exist no canonical shortest path of the form $(a_1, \ldots, a_i, \ldots, a_j, \ldots, a_k, \ldots)$, where $a_i, a_j, a_k \in A$ and the search level of a_j is lower than the level of both a_i, a_k; besides, each arc's search level is always lower or equal to that arc's maximum level, which is computed during the hierarchy construction phase and is equal to the maximum level l such that the arc belongs to G^l. The speed of the query is due to the fact that the search scopes occur mostly on a high hierarchy level, with fewer arcs and nodes than in the original graph. A heuristic extension of the HH algorithm to dynamic static graphs with a detailed experimental evaluation can be found in (109).

Hierarchy construction. As the construction of the highway hierarchy is the most complicated part of HH algorithm, and also the most interesting to gain insight on how the algorithm works, we endeavour to explain its main traits in more detail. For simplicity, in this paragraph we will assume that the graph is undirected; therefore, we will denote the set of edges by E. An extension to directed graphs is easy to derive (125; 109). Given a local extensionality parameter H (which measures the degree at which shortest path queries are satisfied without stepping up hierarchical levels) and the maximum number of hierarchy levels L, the iterative method to build the next highway level $l+1$ starting from a given level graph G^l is as follows:

1. For each $v \in V$, build the neighbourhood $N_H^l(v)$ of all vertices reached from v with a simple Dijkstra search in the l-th level graph up to and including the H-st settled vertex. This defines the local extensionality of

each vertex, i.e. the extent to which the query "stays on level l".

2. For each $v \in V$:

   (a) Build a partial shortest path tree $B(v)$ from v, assigning a status to each vertex. The initial status for v is "active". The vertex status is inherited from the parent vertex whenever a vertex is reached or settled. A vertex w which is settled on the shortest path $(v, u, \ldots, w)$ (where $v \neq u \neq w$) becomes "passive" if

   $$|N_H^l(u) \cap N_H^l(w)| \leq 1. \tag{1.1}$$

   The partial shortest path tree is complete when there are no more active reached but unsettled vertices left.

   (b) From each leaf t of $B(v)$, iterate backwards along the branch from t to v: all arcs (u, w) such that $u \notin N_H^l(t)$ and $w \notin N_H^l(v)$, as well as their adjacent vertices u, w, are raised to the next hierarchy level $l + 1$.

3. Select a set of *bypassable* nodes on level $l + 1$; intuitively, these nodes have low degree, so that the benefit of skipping them during a search outweights the drawbacks (i.e., the fact that we have to add shortcuts to preserve the algorithm's correctness). Specifically, for a given set $B^{l+1} \subset V^{l+1}$ of bypassable nodes, we define the set S^{l+1} of shortcut edges that bypass the nodes in B^{l+1}: for each path $p = (s, b_1, b_2, \ldots, b_k, t)$ with $s, t \in V^{l+1} \setminus B^{l+1}$ and $b_i \in B^{l+1}, 1 \leq i \leq k$, the set S^{l+1} contains an edge (s, t) with $c(s, t) = c(p)$. The core $G_C^{l+1} = (V_C^{l+1}, E_C^{l+1})$ of level $l + 1$ is defined as: $V_C^{l+1} = V^{l+1} \setminus B^{l+1}$, $E_C^{l+1} = (E^{l+1} \cap (V_C^{l+1} \times V_C^{l+1})) \cup S^{l+1}$.

The result of the contraction is the contracted highway network G_C^{l+1}, which can be used as input for the following iteration of the construction procedure. It is worth noting that higher level graphs may be disconnected even though the original graph is connected.

1.4.4.2 Dynamic Node Routing

Separator-based multi-level methods for the SPP have been used by many authors; we refer to (81) for the basic variant. The main idea behind separator-based methods is to define, given a subset of the vertex set $V' \subset V$, the shortest path overlay graph $G' = (V', A')$ with the property that A' is a minimal set of edges such that $\forall u, v \in V'$ the shortest path length between u and v in G' is equal to the shortest path length between u and v in G. In other words, there is an arc $(u, v) \in A'$ if and only if for any shortest path from u to v in G then no internal node of the paths (i.e. all nodes except u and v) belongs to V'. It can be shown that G' is unique (81). Usually, the set of separator nodes V' is chosen in such a way that the subgraph induced by $V \setminus V'$ consists of small components

of similar size. In a bidirectional query algorithm, the components containing source and target node are wholly searched, but starting from the separator nodes only edges of the overlay graph G' are considered. This approach can be generalized and applied in a hierarchical way, building several levels of overlay graphs with node sets $V = V_0 \supseteq V_1 \supseteq \cdots \supseteq V_L$ so that the following property is mantained: $\forall l \leq L-1$, for all node pairs $s,t \in V_l$ the part of the shortest path between s and t that lies outside the level l components to which s and t belong is entirely included in the level $l+1$ overlay graph. As the overlay graphs become smaller with the increasing level in the hierarchy, a shortest path computation becomes faster because most of the search for a long-distance s,t path takes place on the highest hierarchy level, and thus fewer nodes are explored. A path on the original graph can then be reconstructed, because each arc at level l has a unique decomposition as level $l-1$ arcs.

In (126), an arbitrary subset $V' = V'(V)$ of V is considered instead of separator nodes; in practice, the set is chosen in such a way that it contains the most important nodes, i.e. those that appear "more often" on shortest paths. This yields a smaller set V', more uniformly distributed over the whole graph, and thus G' will be smaller, resulting in a smaller space consumption and a faster query algorithm. However, since in this case $V \setminus V'$ is no longer made of small isolated components, the query algorithm is not as simple as in canonical separator-based methods. From a theoretical point of view the same principle holds: we might want to explore nodes from source and target until the queue in Dijkstra's algorithm only contains nodes that are covered by V' (i.e. there is at least one node $v \in V'$ on the shortest path from the root to any leaf of the current partial shortest path tree), and then switch to the overlay graph G', or to a higher level in the overlay graph hierarchy in the case of a multi-level approach. This, however, does not yield good results in practice, because we cannot tell in advance how many nodes we will have to explore until the whole partial shortest path tree is covered by V'. The main challenge is therefore to compute the set of all covering nodes for the partial shortest path tree T rooted at s as quickly as possible.

Many possible strategies are suggested in (126), including an aggressive variant which stops the search whenever a node in V' is encountered, and which yields a superset of the covering nodes. Some other analysed possibilities may require a greater computational effort, while finding the minimal set of covering nodes. Once the set (or a superset) of all covering nodes for a given level of the overlay graph has been computed, the search can switch to the next level, until the shortest path is found, which is guaranteed to happen at the topmost level. The choice of level node sets $V = V_0, V_1, \ldots, V_L$, where $V_i = V'(V_{i-1})$ for all $i > 0$, is critical for query times: these nodes should correspond to nodes that appear very often on shortest paths, i.e. road network junctions with high-importance, such as highway access points. The Highway Hierarchies algorithm (Section 1.4.4.1) is employed in (126) to choose the node sets.

The main advantage of this approach is that overlay graphs can be computed in a very short time because they only require the application of Dijkstra's algorithm on limited parts of the graph; besides, if a few arc costs change there is no need to recompute the whole overlay graphs, but only a small part of them — the part which is affected by the change. Certainly, if the changed arc does not belong to the partial shortest path tree of a given node, the construction phase from that node need not be repeated. In particular, during the pre-processing phase, we can build for each node v a list of all nodes that can be affected if the cost of one of the outgoing arcs from v changes. If these lists are kept in memory, then one knows exactly which parts of the overlay graphs are affected by the change and must be recomputed. The construction phase is repeated only when necessary. After the update step the bidirectional query algorithm correctly computes shortest paths.

1.4.4.3 Contraction Hierarchies

One of the main concepts used in the Highway Hierarchies algorithm (Section 1.4.4.1) is that of *contraction*: bypassable nodes are selected and then iteratively removed from the input graph, adding *shortcuts* (i.e. new arcs) in order to preserve distances with respect to the original graph. The Contraction Hierarchies algorithm (64) is a speedup technique for Dijkstra's algorithm on static graphs which is based solely on contraction: all nodes are ordered by some importance criterion, and then a hierarchy is generated by iteratively contracting the least important node. Thus, the query algorithm is extremely simple: a bidirectional Dijkstra search is started, where the forward search from the source only relaxes arcs leading to more important nodes, while the backward search from the destination only relaxes arcs coming from more important nodes. The Contraction Hierarchies algorithm is remarkably simple, yet it is very effective, yielding large speedups with respect to Dijkstra's algorithm and a very small space occupation of the preprocessed data. Indeed, if one is interested only in computing shortest path distances (i.e. the sequence of edges that form the shortest path on the original graph is not needed), then the preprocessed graph occupies less space than the original input (64).

Clearly, the node ordering criterion is the crucial part of the algorithm, as it determines the order in which nodes are contracted and thus the quality of the final hierarchy. (64) analyses several different criteria and their combination. One of the most important factors in computing a node's importance is the arc difference, i.e. the number of shortcuts that would be created if the node is bypassed minus the number of incoming and outgoing arcs of that node. This quantity has an influence on both the space overhead and query performance; it is easy to see that nodes with small arc difference are more appealing for contraction, as their removal yields a graph with a smaller number of arcs. Another important factor is uniformity: it seems to be a good idea to contract

nodes everywhere in the graph, instead of repeatedly contracting nodes in the same small region. Other tested criteria include estimations of the contraction cost and of query performance. Combinations of several criteria with different weights are possible. Since the contraction of a node may influence the node ordering of neighbouring nodes, three strategies are tested:

1. the priority of adjacent nodes is recomputed after each contraction step,
2. the priority of all nodes is recomputed periodically,
3. the priority of the node chosen for contraction is recomputed *before* the contraction, so that if its priority increases and the node is no longer the minimum element then it is skipped and reinserted into the priority queue with the new value.

An estension of Contraction Hierarchies to the time-dependent case is described in (17); the authors report the main ideas, but there are no computational experiments. Therefore, it is difficult to analyse the performance of an approach based on contraction only in practice. Interestingly, the ideas in (17) are similar to those described in Chapter 4 of our work, although they have been developed indipendently. We discuss some of the problems that may have arised during the implementation of time-dependent Contraction Hierarchies in Section 4.4.

1.4.5 Goal-directed search: A^*

A^* (80) is an algorithm for goal-directed search which is very similar to Dijkstra's algorithm (Section 1.4.2). The difference between the two algorithms lies in the priority key. For A^*, the priority key of a node v is made up of two parts: the length of the tentative shortest path from the source to v (as in Dijkstra's algorithm), and an underestimation of the distance to reach the target from v. Thus, the key of v represents an estimation of the length of the shortest path from s to t passing through v, and nodes are sorted in the priority queue according to this criterion. The function which estimates the distance between a node and the target is called potential function π; the use of π has the effect of giving priority to nodes that are (supposedly) closer to target node t. If the potential function is such that $\pi(v) \leq d(v,t)\ \forall v \in V$, where $d(v,t)$ is the distance from v to t, then A^* always finds shortest paths (80); otherwise, it becomes a heuristic. A^* is equivalent to Dijkstra's algorithm on a graph where arc costs are the reduced costs $w_\pi(u,v) = c(u,v) - \pi(u) + \pi(v)$ (82). From this, it can be easily seen that if $\pi(v) = 0\ \forall v \in V$ then A^* explores exactly the same nodes as Dijkstra's algorithm, whereas if $\pi(v) = d(v,t)\ \forall v \in V$ only nodes on the shortest path between s and t are settled, as arcs on the shortest path have zero reduced cost. A^* is guaranteed no more nodes than Dijkstra's algorithm. In particular, if

$\pi(v)$ is a good approximation from below of the distance to target, A^* efficiently drives the search towards the destination node, i.e. the search space is not a circle centered at s, but an ellipse directed towards t (see Figure 1.4). A^* can be easily applied on a time-dependent graph with the FIFO property, as long as the potential function $\pi(v)$ is a valid lower bound for $d(v,t,\tau)\ \forall\tau \in T$; an analysis of this scenario can be found in (32).

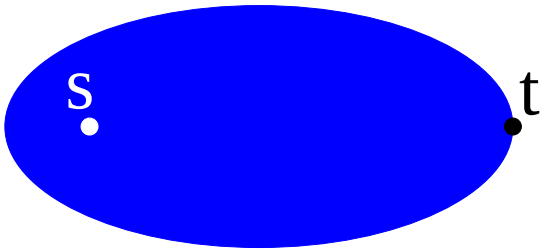


Figure 1.4: Schematic representation of A^* algorithm search space

1.4.5.1 The ALT algorithm

On a road network, Euclidean distances can be used to compute the potential function, possibly dividing by the maximum allowed speed if arc costs are travelling times instead of distances. This obviously holds true even for the time-dependent case. On static graphs, a significant improvement over Euclidean potentials can be achieved using *landmarks* (67). The main idea is to select a small set of nodes in the graph, sufficiently spread over the whole network, and precompute all distances between these nodes (which we call landmarks) and any node of the vertex set. Then, by triangle inequalities, it is possible to derive lower bounds to the distance between any two nodes. Suppose we have selected a set $L \subset V$ of landmarks, and we have stored all distances $d(v,\ell), d(\ell,v)\,\forall v \in V, \ell \in L$; the following triangle inequalities hold: $d(u,t) + d(t,\ell) \geq d(u,\ell)$ and $d(\ell,u) + d(u,t) \geq d(\ell,t)$. Therefore $\pi_f(u) = \max_{\ell\in L}\{d(u,\ell) - d(t,\ell), d(\ell,t) - d(\ell,u)\}$ is a lower bound for the distance $d(u,t)$, and it can be used as a valid potential function for the forward search (67). Bidirectional search can be applied: a forward search is started on G from the source using a potential function π_f which estimates the distance to reach the target, and a backward search is started on the reverse graph $\overline{G}$ from the destination using a potential function π_b which estimates the distance to reach the source (on $\overline{G}$). The two potential function must be consistent (68), which means that $\forall(u,v) \in A$ the reduced cost for the forward search $w_{\pi_f}(u,v)$ on G is equal to the reduced cost for the backward search $w_{\pi_b}(v,u)$ on $\overline{G}$. This translates to $\pi_f(v) + \pi_b(v) = \kappa\ \forall v \in V$ for some constant κ.

Bidirectional A^* with the potential function described above is called ALT;

an experimental evaluation on static graphs can be found in (68). It is straightforward to observe that, if arc costs can only increase with respect to the original value used to compute distances to and from landmarks, then the potential function associated with landmarks yields valid lower bound, even on a time-dependent graph. In (46) this idea is applied to a real road network in order to analyse the algorithm's performance both in the case of arc cost updates and of time-dependent cost functions, but in the latter scenario the ALT algorithm is applied in an unidirectional way.

The size of the search space greatly depends on how landmarks are positioned over the graph, as it severely affects the quality of the potential function. Several heuristic selection strategies have been proposed; there is usually a trade off between preprocessing time and quality of the landmark choice. So far no optimal strategy with respect to random queries has been found, i.e. no strategy guarantees to yield the smallest search spaces with respect to shortest path computations where source and destination nodes are chosen at random. Commonly used selection criteria are *avoid* and *maxCover* (70).

1.4.6 The SHARC algorithm

The SHARC algorithm (18), which employs a multi-level partition approach combined with goal-directed search via arc flags (106), allows fast unidirectional shortest path calculations in large scale networks; it has been recently extended in (41) to compute optimal paths even on time-dependent graphs, and represents the fastest known algorithm so far for exact time-dependent shortest paths computations.

SHARC is a clever combination of well known techniques for the SPP on static graphs, which are then extended to the time-dependent case (41). In particular, SHARC is largely based on the concept of arc flags (106): the graph is partitioned into *cells* $C_0, \ldots, C_k$ and each arc is attached to a sequence of k bits such that if there is a shortest path from u to any node $w \in C_i$ which starts with arc (u,v) then the i-th flag of $(u,v) \in A$ is set to 1. This is a necessary condition; arc flags ensure correctness if more flags than strictly necessary are 1, i.e. there are false positives, but in this case the size of the search space may increase. In the time-dependent case, the i-th flag of (u,v) is 1 if there is a shortest path from u to any node $w \in \mathcal{C}_i$ which starts with (u,v) for at least one departure time $\tau \in \mathcal{T}$. This idea can be extended to a multi-level partition: we consider a family of partitions $\mathcal{C}^0, \ldots, \mathcal{C}^l$ such that for each $i < l$ and $C_n^i \in \mathcal{C}^i$ a cell $C_m^{i+1} \in \mathcal{C}^{i+1}$ exists with $C_n^i \subset C_m^{i+1}$. In this case we say that C_m^{i+1} is the supercell of C_n^i, and we define the supercell of any level l cell as the whole vertex set V. Shortest paths can then be computed with a modified Dijkstra's algorithm: whenever a node v is settled, let i be the lowest level on which both v and the target t are in the same supercell, i.e. v and t are in different level i cells but they are in the same level $i+1$ supercell. Then, when relaxing arcs from a node v, the query

algorithm relaxes only those which have the level i arc flag corresponding to the target cell (i.e. the cell which contains the target) set to 1. The advantage of a multi-level partition is that using the same number of arc flags we can allow a larger number of cells, thus the cells at the lowest level are smaller, leading to a reduced search space. The partition is computed by local optimization of an initial partition obtained from SCOTCH (121); in order to yield good speedups, a partition should fulfill several requirements: the cells must be connected, the size of the cells should be balanced, and the number of boundary nodes should be low.

The basic concept of arc flags is then augmented introducing shortcuts, i.e., by iteratively removing nodes and adding new arcs to preserve distances. In the time-dependent case, distances are preserved by computing each shortcut's cost function with a label-correcting algorithm (Section 1.4.3) between the shortcut's endpoints. The selection criteria for removed nodes are similar to those described for Contraction Hierarchies (Section 1.4.4.3). Additionally, the contraction routine takes into account a hop-limit, i.e. maximum number of arcs of the original graph that a shortcut can represent, and, in the time-dependent case, a limit on the number of interpolation points of each shortcut. The latter issue will be discussed in more detail in Chapter 4. Inserted shortcuts have their arc flags set automatically. The preprocessing phase of SHARC has several successive phases that try to refine arc flags: the number of false positive flags is decreased so as to reduce the search space. Also, some cells are pruned: if one cell has all its neighboring cells contained in the same supercell, then all arcs inside the cell which do not have any flag set to 1 (except the flag corresponding to the cell to which they belong, the so called own-cell flag) can be deleted from the graph used as input for the preprocessing of higher levels of the hierarchy.

An interesting application of SHARC is the multi-metric query scenario, that is, a scenario where shortest paths queries are run on the same graph but can use one of several available cost functions (metrics), instead of only one as we usually assume. To this end, the graph partition is the same for all metrics, but arc flags are computed independently for each one of them. In a final preprocessing step, all arc flags are merged with a bitwise binary OR operation, so that a flag is set to 1 if it is 1 for at least one metric. This idea is based on the assumption that the metrics are related, therefore the number of nonzero arc flags should not increase by a large amount during the final merging.

The SHARC algorithm has been developed to yield a very large speedup when applying unidirectional search. As such, it is applicable to those scenarios where bidirectional search is typically not possible, including time-dependent networks. Note that on static graphs SHARC can be used in a bidirectional manner, yielding even larger speedups. Originally, SHARC allowed only suboptimal queries with no appproximation guarantee on time-dependent graphs (18); however, it could be used in an exact way to compute shortest path on time-

expanded timetable information networks (see (123) for a discussion on efficient ways to model timetable information with graphs). Later, in a very recent paper at the time of writing this thesis (41) SHARC has been extended to compute optimal solutions even on time-dependent networks, and currently represents the state-of-the-art of time-dependent shortest path algorithms, with average query time of a few dozens millliseconds on continental sized road networks. However, due to the intensive use of arc flags, SHARC does not work in a dynamic scenario: whenever an arc cost function changes, arc flags should be recomputed, even though the graph partition need not be updated. Moreover, the preprocessing phase takes several hours on large-scale road networks.

1.5 Contributions

From a theoretical point of view, the TDSPP on FIFO networks is well solved by Dijkstra's algorithm (see Section 1.4.2 for a description of the algorithm). However, if the road network is very large then Dijkstra's algorithm may be too slow for several interesting industrial applications that need point-to-point computations. Moreover, the practical applications that we described in Section 1.1 have some characteristics that are worth underlining.

- The application has to answer all shortest paths queries on the same road network: unlike other applications, in this case there is a fixed network which is given as input, and then only arc costs can possibly change. Thus, we have to repeatedly and quickly solve similar problems on the same input. However, shortest paths trees reoptimization techniques (119; 120) are not effective in this case, as the source and destination nodes may be positioned in completely different parts of the network in consecutive queries. Therefore it makes sense to invest a large amount of resources (computational time and memory space) in a preprocessing phase which serves the purpose of speeding up all successive shortest paths computations. As the input network does not change, this preprocessing phase can be done once and for all.

- Dijkstra's algorithm computes more shortest paths than needed: indeed, at its termination we have the shortest path from the source node to all other nodes in the graph, even though early stopping criteria exist. However, for point-to-point shortest paths computations only *one* path need be computed, that is the shortest path between the source and the destination. An efficient algorithm should try to disregard other paths as much as possible, and focus on the fast computation of the desired answer.

- Road networks have structural similarities: although road networks with travel times as arc weights are not necessarily Euclidean, they still share a

hierarchical structure, so that there are local roads and progressively more important roads such that important roads appear more often on long shortest paths. This is easy to see if we consider the real-world networks that are represented: for long distance travels, the shortest (i.e. fastest) path will almost always pass through motorway segments. Besides, road networks are very sparse: the average degree of a node is usually between 2 and 3. This structural properties could be exploited to develop ad-hoc algorithms.

A large number of very fast algorithms has been developed for point-to-point shortest path computations on static graphs using the hierarchical structure of road networks and by directing the search towards the destination (see Section 1.4). The main problem for their application on time-dependent graphs is that these techniques are intrinsically bidirectional, i.e. they start a search from both the source and destination until the two search scopes meet. Bidirectional search is not only useful because it cuts down the search space (Section 1.4.2), but it is sometimes *necessary* in order for hierarchical speedup techniques to work (Section 1.4.4). Consider a road network with an associated road hierarchy (we will not give a formal definition here — we rely on intuition), and suppose that we want to compute the shortest path between two distant nodes. Intuitively, it is reasonable to assume that the shortest path search can be restricted to the highest level of the hierarchy when we are "in the middle" of the path. However, while switching to a higher level in the hierarchy is easy, descending to a lower level is more difficult. We give a real-world example to clarify this concept. When driving a car to a distant destination, as soon as we meet an access point to the highway network we will usually want to enter it, i.e. we switch to a higher level in the hierarchy. Then we stay on the highway network until we reach an exit point near to the destination; at that point we get off the highway network and continue our path on local roads. In order to identify the exit/entrance point of the highway network which is closest to the destination we implicitly start a backward search from the target, which allows us to determine where we have to leave the highway network, i.e. descend to a lower level of the hierarchy. Thus, the algorithm is bidirectional. Given a source/destination pair, on a static graph the shortest path between the two points is unique, therefore the entrance points to the highway network that appear on the shortest path are fixed; however, this is not the case on time-dependent graphs, as the shortest path depends on the departure time. Moreover, a backward search from the destination cannot be started as we do not know which time instant should be used to compute the time-dependent arc costs, the arrival time at the destination being unknown (we would need the optimal solution for this).

In this work we propose a general approach for bidirectional search on large-scale time-dependent graphs. We explain the main ideas and prove that the algorithm is correct if used in conjunction with Dijkstra's algorithm or A^*. The al-

gorithm is based on running a time-*independent* search from the destination in order to bound the set of nodes that have to be explored by the time-dependent forward search from the source. We discuss several theoretical improvements of the basic idea: in particular, we study methods to improve the bounds used by the backward search in order for it to terminate faster. An experimental evaluation shows that this idea alone is able to cut down the search space. Moreover, our bidirectional algorithm allows us to easily generalize the concepts used for hierarchical speedup techniques on static graphs, so that we are able to efficiently build a two-levels hierarchy on the time-dependent road network. The use of a hierarchy yields a large improvement in query times, and also required preprocessing time and space decrease. Our method is developed with the dynamic time-dependent scenario in mind; thus, we propose an approach which allows the time-dependent arc cost functions to change, while requiring only a small computational overhead in order to restore optimality of the hierarchy. We analyse the experimental performance of the proposed method in several respects, and show that it yields an improvement over other known speedup techniques. In particular, our implementation of this algorithm is the first which is capable of dealing with the dynamic time-dependent scenario while having average query times in the order of a few milliseconds. If we are willing to accept a small approximation constant, it is also the fastest known method for time-dependent large-scale road networks (i.e. not necessarily dynamic) at the moment of writing this thesis. We discuss the practical integration of an efficient C++ implementation of the proposed algorithm within an existing industrial C# platform that manages a path computing web service, receives real-time traffic updates and deals with traffic forecastings.

The TDSPP can also be modeled through a mathematical programming formulation, which allows for more freedom in choosing the cost functions (which may become nonlinear) and for dropping the assumption of the FIFO property. Complicating constraints such as prohibited turnings are also easy to take into account within a mathematical formulation. This is useful as a mean to study the behaviour of shortest paths on non-FIFO networks. We propose a mathematical program that models the TDSPP in general non-FIFO networks with arbitraty cost functions. Depending on the expression of the cost function, computing a shortest path reduces to solving a Mixed-Integer Linear Program (MILP) or a Mixed-Integer Nonlinear Program (MINLP). In this work, we study efficient general-purpose algorithms for both classes of problems. We test these algorithms on benchmark instances taken from the literature to assess their quality on a large group of problems, as well as for shortest paths computations.

Within the context of solving MILPs by a Branch-and-Bound algorithm, we propose a new strategy for branching based on a quadratic optimization approach. Computational experiments show that, on the majority of our test instances, this approach enumerates fewer nodes than traditional branching, and

is competitive in terms of speed. On average, the number of nodes in the enumeration tree is reduced by a factor two, while computing time is comparable. On a few instances, the improvements are of several orders of magnitude in both number of nodes and computing time. Moreover, we propose a mathematical program that models the problem of finding a split disjunction closing a large integrality gap at each node of a branch-and-cut algorithm. This formulation is a MILP for which we find feasible solutions using general heuristics and a local branching algorithm. Although an implementation of this approach is not competitive in practice with the quadratic optimization method discussed above, because of the required computational effort, we believe that it is interesting for future research.

Finally, we present a general-purpose heuristic for MINLPs based on Variable Neighbourhood Search, Local Branching, Sequential Quadratic Programming and Branch-and-Bound. We test the proposed approach on the MINLPLib, discussing optimality, reliability and speed. It turns out that our method, acting on the whole MINLPLib, is able to find optima equal to or better than those reported in the literature for 55% of the instances, ranking first under this respect among known methods. The closest competitor is a branch-and-bound approach (SBB + CONOPT) that finds putative global optima for 37% of the testset. We improve the best known solutions in 7% of the cases.

1.6 Overview

The rest of this work is organized as follows.

In Part I we present novel combinatorial approaches to the solution of the time-dependent shortest path problem. In Chapter 2 we discuss a method based on defining guarantee regions for the shortest path computations, that is, small subgraphs that can be quickly explored and provide an approximation guarantee. In Chapter 3 we describe the main ideas for bidirectional search on time-dependent graphs, prove correctness of our approach and discuss several theoretical improvements that are important for the practical implementation. In Chapter 4 we introduce a two-levels hierarchical setup called *core routing* for our main search algorithm, discuss methods to restore optimality of the hierarchy in the dynamic scenario where arc cost functions are allowed to change, and study the difficulties of a multi-level hierarchical approach. In Chapter 5 we give a detailed experimental evaluation of both Dijkstra's algorithm and A^* within the bidirectional search approach that we presented. We also test the dynamic scenario and analyse the tradeoff between query speed and hierarchy update speed. Finally, in Chapter 6 we discuss the integration of our efficient implementation of the bidirectional search algorithm with an existing real-world industrial application which gathers traffic information and provides a path computing web service.

In Part II we discuss general-purpose solution methods for mathematical programs with integer variables, such as the mathematical programming formulation of the TDSPP that we discussed in Section 1.3.2. We focus on two classes of problems: Mixed-Integer Linear Programs and Mixed-Integer Nonlinear Programs. In Chapter 7 we propose new strategies for branching within the context of solving MILPs by a a Branch-and-Bound algorithm; we propose a mathematical formulation for the problem of finding a split disjunction closing a large integrality gap at each node of the Branch-and-Bound tree, and a fast heuristic strategy for the same problem, discussing computational results. Chapter 8 presents a general-purpose heuristic for MINLPs based on Variable Neighbourhood Search, Local Branching, Sequential Quadratic Programming and Branch-and-Bound. We provide extensive computational experiments on a set of benchmark instances to show that our method is very reliable in terms of the quality of the solution found, and also improves the best known solutions in some of the cases. Chapter 9 reports computational results for the solution of the time-dependent shortest path problem with the mathematical formulation based methods that we presented.

Finally, Chapter 10 draws the conclusions of this work and discusses future research.

Part I

Combinatorial Methods

On FIFO networks, Dijkstra's algorithm (Section 1.4.2) is typically the algorithm of choice for the solution of the TDSPP. Dijkstra's algorithm is a fully combinatorial method, in the sense that it does not work with a mathematical programming representation of the problem, but tries to construct the optimum by explicitly exploring the graph on which it is applied. As discussed in Chapter 1, Dijkstra's algorithm is too slow for our needs; therefore, we would like to develop more efficient methods. The main purpose of this part is to devise algorithms which explore fewer nodes than Dijkstra's algorithm; that is, combinatorial methods that work on the graph representation, and "move" from node to node touching the smallest possible number of vertices until the shortest path between the source and the destination is found. This scenario is made difficult by the fact that arc costs are time-dependent.

At the beginning of this thesis, no efficient speedup techniques for the TDSPP were known, although there was a lot of ongoing work on static graphs (see Section 1.4); we believed that fast computations for the TDSPP were possible only if we accepted approximated solutions. In this chapter we will present our work in chronological order, to see how we developed an *exact* algorithm for the TDSPP that deals very efficiently with dynamic scenarios. The first approach that we propose (Chapter 2) is based on computing approximated solutions to the TDSPP, by exploring pre-computed subgraphs instead of the full network. Its theoretical description provides interesting idea, but the method's implementation suffers from a too large memory occupation when applied to continental sized road networks. However, some of the basic ideas are behind our framework for bidirectional search presented in Chapter 3. The extension of bidirectional search from static graphs to time-dependent graphs is a very useful tool, that yields a reduction of the search space, as well as the means necessary to apply hierarchical speedup techniques on time-dependent networks. We will discuss hierarchical methods on time-dependent graphs in Chapter 4. The detailed experimental evaluation provided in Chapter 5 assesses the usefulness of our approach. Finally, in Chapter 6 we present a real-world industrial application of our algorithm, as one of the main components of the website of the Mediamobile company.

Chapter 2

Guarantee Regions

Consider an application scenario with a server machine that answers shortest paths queries provided by connected clients; this is exactly the type of application that we had in mind for this thesis. In this case, assuming that the average number of shortest paths queries that have to be answered in a given time interval is known, we would like to guarantee that each computation can be carried out in the allotted time frame. This motivates our study of an algorithm capable of providing an upper bound on computational times. In this section, we propose a method with a preprocessing phase that provides an upper bound on the number of nodes that have to be explored during a shortest path computation; this can be translated into an upper bound to the maximum computational time, using an upper bound on the time spent per node and on the time for each priority queue operation. All computations are parameterized by an approximation constant K that determines the quality of the solution with respect to the optimum. By increasing the value of the approximation constant that is used throughout the whole method, one is able to decrease this upper bound on the number of explored nodes (up to a certain degree), so that the desired time requirements can be met.

The rest of this section is organized as follows: in Section 2.1 we define a guarantee region with an approximation property for the point-to-point shortest path problem on an unclustered graph; then we extend those concepts to a clustered graph, so as to make them useful in practice. In Section 2.2 we describe the preprocessing algorithm, whereas in Section 2.3 we describe the query algorithm. In Section 2.4, we discuss practical issues, such as how to effectively store guarantee regions, give some computational results, and analyze the drawbacks of this method.

2.1 Definitions and main ideas

Given a source node s and a destination node t, the main idea behind guarantee regions is to compute the shortest path p^* between s and t on G_ρ, which gives an upper bound on the shortest path cost for that node pair over all possible departure times. By means of this upper bound $\rho(p^*)$ one can determine, in a preprocessing phase, all nodes that have to be explored when computing the shortest path from s to t in order to obtain a K-approximate solution, where K is a given constant. This can be done in polynomial time by identifying all nodes lying on a path p (from s to t) whose λ-weighted cost is strictly lower than $\rho(p^*)/K$. Formally:

Definition 2.1.1. *Let $K > 1$, $s,t \in V$ and p be a path from s to t. We define the guarantee region between s and t as:*

$$\mathcal{R}(K,p) = \left\{ v \in V | (v \in p) \vee (v \in p = (s, v_1, \ldots, v_n, t) : \lambda(p) < \frac{1}{K}\rho(p) \right\}.$$

The approximation guarantee is ensured by the following proposition.

Proposition 2.1.2. *Let $K > 1$, $s,t \in V$, and p^* be the shortest path between s and t on G_ρ. Let r^* be the shortest path between s and t with departure time τ_0 on the graph $G[\mathcal{R}(K,p^*)]$. Then $\gamma(r^*,\tau_0) \leq Kd(s,t,\tau_0)$ for any departure time τ_0.*

Proof. Let q be the shortest path from s to t with departure time τ_0. Suppose $\gamma(r^*,\tau_0) > K\gamma(q,\tau_0)$; therefore, q contains a node $v \notin \mathcal{R}(K,p^*)$. Let q^* be the shortest $s \rightarrow t$ path on $G_\rho[\mathcal{R}(K,p^*)]$. We have the chain $\lambda(q) \leq \gamma(q,\tau_0) < \frac{1}{K}\gamma(r^*,\tau_0) \leq \frac{1}{K}\gamma(q^*,\tau_0) \leq \frac{1}{K}\rho(q^*) \leq \frac{1}{K}\rho(p)$, which implies that all nodes of q are in $\mathcal{R}(K,p^*)$ by definition, including v, which is a contradiction. □

It is straightforward to note that the above proposition holds true in the case where $\mathcal{R}(K,p)$ is defined in terms of a generic $s \rightarrow t$ path, and not necessarily the shortest path in G_ρ. This is because the upper bound on the cost of the shortest path from s to t for any departure time can be obtained from any $s \rightarrow t$ path weighted by ρ. However, it makes sense to choose the shortest path between s and t on G_ρ, as in Definition 2.1.1, so as to minimize $\rho(p)/K$ and have a smaller guarantee region.

Proposition 2.1.3. *Let p^* be the shortest $s \rightarrow t$ path in G_ρ, and p be another (different) $s \rightarrow t$ path. If $p^* \subset \mathcal{R}(K,p)$ then $\mathcal{R}(K,p^*) \subseteq \mathcal{R}(K,p)$.*

Proof. By definition, for all $v \in \mathcal{R}(K,p^*)$ either $v \in p^*$ or there is a path q from s to t such that $v \in q$ and $\lambda(q) < \frac{1}{K}\rho(p^*) \leq \frac{1}{K}\rho(p)$. In the first case $v \in \mathcal{R}(K,p)$ by hypothesis; in the second case $v \in \mathcal{R}(K,p)$ by its own definition. □

Although the result only holds if $p^* \subset \mathcal{R}(K,p)$, Prop. 2.1.3 is useful because it states that choosing the initial path p as the shortest path in G_ρ is a good choice, even if it is not necessarily the best one.

As long as guarantee regions depend on each node pair (s,t), this approach is largely impractical. It turns out, however, that the computations can be carried out for a set of departure nodes and a set of arrival nodes, while still mantaining the desired approximation constant. The only difference is that the guarantee region should be valid for any choice of s and t in the respective sets. Thus, the resulting guarantee region will have to be somehow "larger". In practice, the graph is covered with connected node sets $V_1,\ldots,V_k$, which are called *clusters*, and a central node c_i is defined for each of them.

Definition 2.1.4. *A covering $V_1,\ldots,V_k$ of V is valid if for all $i \leq k$ there is a vertex c_i such that for all other vertices $v \in V_i$ there is a path between c_i and v entirely contained in V_i. We call c_i the center of cluster i.*

For all $i \leq k$ let:

$$\sigma_i = \max_{v \in V_i} \min_{p=(v,\ldots,c_i)} \rho(p)$$

and:

$$\delta_i = \max_{v \in V_i} \min_{p \in (c_i,\ldots,v), p \subset V_i} \rho(p)$$

be, respectively, the cost of the longest shortest path in G_ρ from v to c_i over all $v \in V_i$ and the cost of the longest shortest path in G_ρ entirely contained in V_i from c_i to v over all $v \in V_i$. These values are finite because we assumed that G is strongly connected, and Definition 2.1.4 ensures that a path entirely contained in V_i from the central node c_i to all other vertices in V_i exists.

To define guarantee regions that are valid for any two nodes in the source and destination cluster, we will proceed in the same way as before; in order to compute a valid upper bound on the cost of the shortest path between any node in the source cluster V_i and any node in the destination cluster V_j, we will have to consider not only the cost of a path between the centers of the two clusters, but also the radii σ_i and δ_j.

Definition 2.1.5. *Given a valid covering $V_1,\ldots,V_k$ of V, let $K > 1$ and p be a path from c_i to c_j, $i \neq j \leq k$. We define the guarantee region between V_i and V_j as:*

$$\mathcal{R}_{ij}(K,p) = \Bigg\{ v \in V | v \in p \cup V_j \vee (v \in q = (c_i, v_1, \ldots, v_n, c_j) : \lambda(q) < \frac{1}{K}(\rho(p) + \sigma_i + \delta_j)) \Bigg\},$$

A similar approach is presented in (103): the vertex set is partitioned in clusters, and precomputed cluster distances are used to accelerate a Dijkstra search;

the authors state that their method can be used in a time-dependent scenario, but they do not provide experimental results for this case.

An approximation guarantee is given by Theorem 2.1.6, which is analogous to Prop. 2.1.2, but considers guarantee regions between clusters. One difficulty arises: in this case, we can no longer simply consider the shortest path between two nodes restricted to the corresponding guarantee region; we have to select the departure cluster dynamically, depending on which cluster center is the closest to s at time τ_0.

Theorem 2.1.6. *Given a valid covering $V_1, \ldots, V_k$ of V, let s, t in V, $K > 1$, and let $i = \arg\min_{n=1,\ldots,k}\{d(s, c_n, \tau_0)\}$, $j : t \in V_j$; suppose $i \neq j$. Let p be the shortest path from c_i to c_j in G_ρ; for any node $v \in V$, let q_v be the shortest path from s to v with departure time τ_0, and let r_v be the shortest path from v to t on $G[\mathcal{R}_{ij}(K, p)]$ with departure time $\gamma(q_v, \tau_0)$. Then*

$$\min_{v \in \mathcal{R}_{ij}(K,p)} \{\gamma(q_v + r_v, \tau_0) | \gamma(q_v, \tau_0) \leq d(s, c_i, \tau_0)\} \leq K d(s, t, \tau_0)$$

for any departure time τ_0.

Proof. Let p^* be the shortest $s \to t$ path with departure time τ_0,

$$M = \{v \in \mathcal{R}_{ij}(K, p) | d(s, v, \tau_0) \leq d(s, c_i, \tau_0)\}, \tag{2.1}$$

and note that $c_i \in M$, so $M \neq \emptyset$; suppose

$$\min_{v \in M} \{\gamma(q_v + r_v, \tau_0)\} > K d(s, t, \tau_0) = K\gamma(p^*, \tau_0).$$

Let

$$m = \arg\min_{v \in M} \{\gamma(q_v + r_v, \tau_0)\}, \tag{2.2}$$

and let us define the following paths: q^* the shortest path from s to m with departure time τ_0, r^* the shortest path from m to t in $G[\mathcal{R}_{ij}(K, p)]$ with departure time $\gamma(q^*, \tau_0)$, x^* the shortest path from s to m in G_ρ, y^* the shortest path from m to t in $G_\rho[\mathcal{R}_{ij}(K, p)]$. Note that 2.1 and 2.2 imply that $\gamma(x^*, \tau_0) \leq d(s, c_i, \tau_0) \leq \sigma_i$; then by 2.2 and by optimality of y^* we have $\lambda(p^*) \leq \gamma(p^*, h\tau_0) < \frac{1}{K}(\gamma(q^* + r^*, \tau_0)) \leq \frac{1}{K}(\gamma(x^* + y^*, \tau_0)) \leq \frac{1}{K}(\rho(x^* + y^*)) \leq \frac{1}{K}(\sigma_i + \delta_j + \rho(p))$, which means that every node of p^* is in $\mathcal{R}_{ij}(K, p)$ by Definition 2.1.5. This also implies that $s \in M$, and thus, by optimality of p^*,

$$\min_{v \in M} \{\gamma(q_v + r_v, \tau_0)\} = \gamma(p^*, \tau_0)$$

which is a contradiction. □

Theorem 2.1.6 suggests a query algorithm to compute valid paths between two nodes; the idea is to find, from the source node, the closest cluster center assuming departure time τ_0, and then "hop on" a guarantee region at that

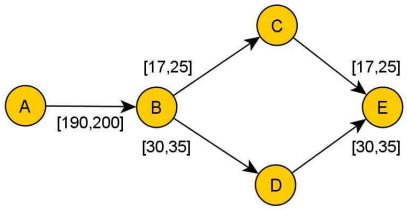


Figure 2.1: The label of each arc (u, v) is $[\lambda(u, v), \rho(u, v)]$.

center. That is, if i is the index of the cluster whose center is the closest to s assuming departure time τ_0, and j is the index of the cluster which contains t, then after settling c_i we constrain the search to explore only nodes in the guarantee region between V_i and V_j. Note that the computed shortest path does not necessarily pass through c_i, but the search is restricted to the guarantee region only after c_i is settled. The query algorithm is described in Section 2.3.

A result similar to Prop. 2.1.3 holds for $\mathcal{R}_{ij}(K, p^*)$ when $p^* \in P^*_\rho(c_i, c_j)$, and serves as a hint to choose our initial path. Unfortunately, guarantee regions defined this way may fail to be minimal.

Proposition 2.1.7. *Given a valid covering $V_1, \ldots, V_k$ of V, for $i, j \leq k, i \neq j$ let p^* be the shortest path between c_i and c_j in G_ρ, and p be another (different) $c_i \to c_j$ path. If $p^* \subset \mathcal{R}_{ij}(K, p)$ then $\mathcal{R}_{ij}(K, p^*) \subseteq \mathcal{R}_{ij}(K, p)$.*

Example 2.1.8. *We give an example of the fact that guarantee regions may fail to have minimal size. We consider $s = A, t = E, K = \frac{6}{5}$ in the graph of Fig. 2.1. Since the shortest path from A to E in G_ρ is $p^* = (A, B, D, E)$, those nodes are included in $\mathcal{R}(6/5, p^*)$. Furthermore, since $\rho(A, B, D, E) = 270$, and since the path $p = (A, B, C, E)$ in G_λ has cost $\lambda(p) = 224 < \frac{5}{6}270 = 225$, we have that $\mathcal{R}(6/5, p^*) = \{A, B, C, D, E\}$. However, it is easy to see that the set $\mathcal{R}' = \{A, B, D, E\}$ has a smaller size and is enough to guarantee the approximation property.*

2.2 Computing the node sets

By Definition 2.1.1 and Definition 2.1.5, it is clear that the most difficult (and expensive) part during the computation of guarantee regions is determining all paths q between s and t with λ-cost $\lambda(q) < H$, where H is a given value. H depends whether we are in the unclustered or in the clustered case. Computing H itself is not computationally expensive: by Definition 2.1.5, we are interested in $H = \frac{1}{K}(\rho(p) + \sigma_i + \delta_j)$ where p is the shortest path between c_i and c_j on G_ρ. Therefore, p can be obtained with an application of Dijkstra's algorithm; similarly, σ_i and δ_j can be calculated by growing shortest path trees from c_i (respectively, c_j) on $\bar{G}_\rho$ (respectively, G_ρ).

We now propose an algorithm to compute the set of all nodes that belong to a path from a node s to a node t with total cost $< H$, adding some lines to Dijkstra's algorithmm and applying it on the reverse graph (note that, with straightforward changes, the same holds on the original graph). Let us define the limited-width shortest paths tree $T_{u,L}$ from a given node u of width L as the shortest paths tree that contains all nodes v such that $d_\lambda(u,v) < L$. We call $\ell[v]$ Dijkstra's algorithm label of a node $v \in T_{s,L}$ computed on the direct graph, i.e. $\ell[v] = d_\lambda(u,v)$, and $\overline{\ell}[v]$ Dijkstra's algorithm label of a node v computed on the reverse graph, i.e. $\overline{\ell}[v] = d_\lambda(v,u)$. We assume to set $\ell[v] = \infty$ if $v \notin T_{s,L}$, and $\overline{\ell}[v] = \infty \, \forall v \in V$. Algorithm 1 computes the desired set.

Algorithm 1 Find all nodes on a path with total λ-cost $< H$ from a node s to a node t

Build $T_{s,H}$ on graph G_λ
$Q \leftarrow \{t\}$
$\overline{\ell}[t] \leftarrow 0$
$S \leftarrow \emptyset$
$E \leftarrow \emptyset$
$stop =$ `false`
if $\ell[t] \neq \infty$ **then**
  while $Q \neq \emptyset \wedge \neg stop$ **do**
    extract $u \leftarrow \arg\min_{q \in Q}\{\overline{\ell}[q]\}$
    $E \leftarrow E \cup \{u\}$
    if $\overline{\ell}[u] + \ell[u] < H$ **then**
      $S \leftarrow S \cup \{u\}$
    if $\overline{\ell}[u] \geq H$ **then**
      $stop =$ `true`
    for all arcs $(v,u) \in A$ **do**
      if $v \notin E$ **then**
        if $v \notin Q$ **then**
          $\overline{\ell}[v] \leftarrow \overline{\ell}[u] + \lambda(v,u)$
          $Q \leftarrow Q \cup \{v\}$
        else if $\overline{\ell}[u] + \lambda(v,u) < \overline{\ell}[v]$ **then**
          $\overline{\ell}[v] \leftarrow \overline{\ell}[u] + \lambda(v,u)$
return S

Proposition 2.2.1. *Algorithm 1 returns all nodes on a path p from s to t such that $\lambda(p) < H$.*

Proof. First, we note that this algorithm is a modification of Dijkstra's algorithm which adds some lines that do not interfere with the correctness of Dijkstra's algorithm; there is one additional terminating condition on the main loop: the algorithm stops if it has settled a node u with $\overline{\ell}[u] > H$.

First part: Algorithm 1 returns all nodes on a path p from s to t such that $\lambda(p) < H$. Suppose there is a node $u \notin S$, $u \in q$ where q is an $s \to t$ path such that $\lambda(q) < H$; since there is a path from s to t with cost $< H$, we have $d_\lambda(u,t) < H$, so node u is scanned because the additional terminating condition on the main loop does not apply. Also, we have that $d_\lambda(s,u) + d_\lambda(u,t) \leq \lambda(q)$ by optimality. Thus, when the node is scanned, the test on line 11 holds since $\ell[u] + \overline{\ell}[u] = d_\lambda(s,u) + d(u,t) \leq \lambda(q) < H$, and u is added to S, which is a contradiction.

Second part: Algorithm 1 returns only nodes on a path p from s to t such that $\lambda(p) < H$. Suppose there is a node $u \in S$ such that $\nexists q = (s, v_1, \ldots, v_n, t)$ such that $\lambda(q) < H, u \in q$. Since $u \in S$, then it has been added on line 12; therefore, $\ell[u] + \overline{\ell}[u] < H$. In this case, by definition of $\overline{\ell}[u]$ and $l[u]$, we can concatenate the shortest paths from s to u and from u to t to build a $s \to t$ path with cost $\ell[u] + \overline{\ell}[u] < H$. The contradiction follows. □

Time requirements for Algorithm 1 are equivalent to applying two times Dijkstra's algorithm on the original graph, thus $O(|A|+|V|\log|V|)$ with Fibonacci's heaps, but each application can be stopped as soon as we reach a distance of H from source node. Hence, effective running time greatly depends on H, which in turns depends on the choice of K and graph's topology: the higher K, the lower the execution time. Space requirements are linear in $|V|$: in addition to Dijkstra's algorithm linear space requirements, we only need to store each node's label in the direct search before applying the reverse search. Note that keeping track of the whole shortest paths trees is not needed.

2.3 Query algorithm

Given a valid covering $V_1, \ldots, V_k$ for V, Theorem 2.1.6 points at a way to compute a K-approximated time-dependent path between any pair of nodes $s, t \in V$. Suppose we have already computed $\mathcal{R}_{ij}(K, p^*)$ $\forall i \neq j, 1 \leq i, j \leq k$, and for p^* is the shortest path between c_i and c_j in G_ρ; let us define $\mathcal{R}_{ii}(K, p^*) = V$ $\forall i, 1 \leq i \leq k$. We will use a slightly modified version of time-dependent Dijkstra's algorithm, where we will call $\ell[v]$ Dijkstra's algorithm label of a node $v \in V$, and we denote by $p[v]$ the parent node for node v. We assume to set $\ell[v] := \infty, p[v] := \texttt{nil}$ $\forall v \in V$. Algorithm 2 respects the theorem's conditions.

Proposition 2.3.1. *Algorithm 2 computes a path p from s to t such that $\gamma(p, \tau_0) \leq K d(s,t,\tau_0)$.*

Proof. First, note that Algorithm 2 is a modification of the time-dependent Dijkstra's algorithm, with an early termination condition which is known to be correct: the algorithm stops as soon as the sink t has been settled, instead of waiting for the queue to be empty. We have to prove that the modifications do not interfere with the algorithm's correctness.

Algorithm 2 Compute a K-approximation of the time-dependent shortest path from a node s to a node t

1: Let $j : t \in V_j$
2: $Q \leftarrow \{s\}$
3: $\ell[s] \leftarrow 0$
4: $S \leftarrow \emptyset$
5: $stop \leftarrow$ `false`
6: $phase \leftarrow 1$
7: $i \leftarrow 0$
8: **while** $\neg stop$ **do**
9: extract $u \leftarrow \arg\min_{q \in Q}\{\ell[q]\}$
10: $S \leftarrow u$
11: **if** $u = t$ **then**
12: $stop \leftarrow$ `true`
13: **if** $phase = 1 \wedge \exists n : u = c_n$ **then**
14: $i \leftarrow n$
15: $phase \leftarrow 2$
16: **for all** arcs $(u, v) \in A$ **do**
17: **if** $phase = 1 \vee v \in \mathcal{R}_{ij}(K, p^*)$ **then**
18: **if** $v \notin S$ **then**
19: **if** $v \notin Q$ **then**
20: $\ell[v] \leftarrow \ell[u] + c((u, v), \tau_0 + \ell[u])$
21: $p[v] \leftarrow u$
22: $Q \leftarrow Q \cup \{v\}$
23: **else if** $\ell[u] + c((u, v), \tau_0 + \ell[u]) < \ell[v]$ **then**
24: $\ell[v] \leftarrow \ell[u] + c((u, v), \tau_0 + \ell[u])$
25: $p[v] \leftarrow u$
26: **return** $t, p[t], p[p[t]], \ldots, s$

As long as $phase = 1$, the algorithm is exactly the same as Dijkstra's algorithm; the critical point is the assignment $phase \leftarrow 2$ on line 15. The test on line 13 is verified as soon as the closest cluster center from the source node s is settled; at this point, the index i is set to be the index of that cluster, and $phase \leftarrow 2$. If $i = j$ it is easy to verify that the algorithm is exactly the same as Dijkstra's algorithm because the test on line 17 is always true for $\mathcal{R}_{ii}(K, p^*) = V$, therefore Algorithm 2 computes the shortest path. Otherwise, when $phase \leftarrow 2$ all nodes in the set M of Theorem 2.1.6 have already been explored and thus added to the queue, and for each of those the shortest time-dependent path from the source has already been computed via Dijkstra's algorithm. Since $phase = 2$, the test on line 17 ensures that only nodes belonging to the guarantee region $\mathcal{R}_{ij}(K, p^*)$ are explored (i.e. added to the queue); this means, by Dijkstra's algorithm correctness, that we are computing shortest time-dependent paths restricted to that

node set. However, these are exactly the conditions of Theorem 2.1.6, and thus when node t is settled we have computed a path which is the optimum of

$$\min_{v \in \mathcal{R}_{ij}(K,p)} \{\gamma(q_v + r_v, \tau_0) | \gamma(q_v, \tau_0) \leq d(s, c_i, \tau_0)\} \leq K d(s, t, \tau_0)$$

for any departure time τ_0. □

In order to provide an upper bound on the computational time of each shortest path computation, we have to provide an upper bound on the number of nodes that are explored. The required upper bound on computational time can then be derived considering the maximum time spent per node (i.e. while settling the node with maximum degree in the graph) and the maximum time for a priority queue operation. It is straightforward to note that, once Algorithm 2 has switched to phase 2, then the number of nodes that can be explored is bounded from above by $|\mathcal{R}_{ij}(K, p^*)| + |V_j|$, where i and j are, respectively, the index of the source and of the destination cluster. We have to provide a bound on the number of nodes explored before switching to phase 2: in order to do so we note that, if we restrict the algorithm in phase 1 to explore only nodes within V_i, where $s \in V_i$, then the approximation guarantee is still valid, although the solution quality may decrease. Thus we require that, if b nodes have already been explored in phase 1, then the algorithm is restricted to explore only nodes in V_i, until it switches to phase 2. It is easy to prove correctness of this approach. An upper bound on the number of explored nodes is then

$$b + |V_i| + |\mathcal{R}_{ij}(K, p^*)| + |V_j|. \tag{2.3}$$

The size of $\mathcal{R}_{ij}(K, p^*)$ can be decreased by increasing K.

2.4 Implementation

To validate the practical usefulness of this approach, we implemented it in the C++ programming language. It turns out that performance is not fully satisfying. In this section we report our results.

2.4.1 Storing node sets

Once all guarantee regions have been computed with the algorithm described in Section 2.2, we have to store them efficiently in memory for a fast access. This issue is crucial for performance, since the query algorithm has to test, for each node, whether it belongs to a given guarantee region or not, and thus the algorithm's efficiency depends on how quickly this answer can be given and how accurately node sets are stored. Assuming that we know each node's position

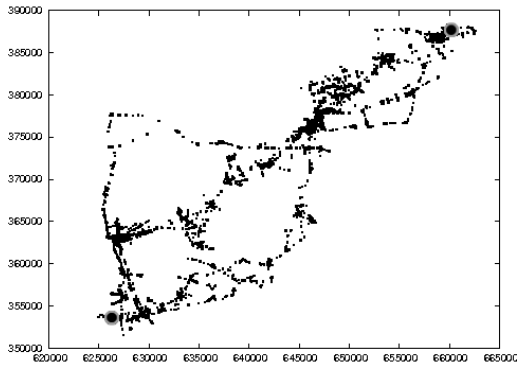


Figure 2.2: Graphical representation of a guarantee region on a plane. Gray circled dots represent source and destination node, while small black square dots represent nodes within the guarantee region.

on a plane, a natural way to store node sets would be to define a geometric container for each guarantee region, e.g. an ellipse; however, with this approach the routine which tests if a node belongs to a given guarantee region yields too many false positive answers, which is due to the fact that guarantee regions are an union of paths, and thus their shape is not necessarily easy to model (see Figure 2.2). Our approach to solve this problem is to associate, with each node, a bit table, or *bit flags*, which are used to determine if a node belongs to the guarantee region between clusters i and j for given $i \neq j$. Suppose we have covered V with k clusters $V_1, \ldots, V_k$; then we associate a table T of $k \times (k-1)$ bits each with each node v, with the property that the j-th bit of the i-th row of T is 1 if and only if: $v \in \mathcal{R}_{ij}(K, p^*)$ if $j < i$, or $v \in \mathcal{R}_{i(j+1)}(K, p^*)$ if $j \geq i$. Since $\mathcal{R}_{ii}(K, p^*) = V$, the corresponding information does not have to be stored, thus each row can have only $k-1$ elements.

2.4.2 Computational analysis

To validate our approach with a prototype, we used a subgraph of France's road network, corresponding to Île-de-France (i.e. Paris and surroundings), This subgraph has ≈ 400000 vertices and ≈ 800000 arcs. Time-dependent costs were modeled as piecewise linear functions of time (expressed in seconds); that is, on each edge we stored 24 breakpoint values, one for each hour over a day, and the arc cost for a given second τ was computed via a linear interpolation of the breakpoints preceding and following τ. For a subset of arcs (8374 arcs in total, all of them corresponding to highways or high importance roads) we used real historical data to compute the breakpoint values for weekdays (see Section 5.1.1); for all remaining arcs we generated breakpoint values using the traffic-free speed value for that arc over a day, and then generating two bend-

ings in the speed profile so as to slow the arc down by a factor of 1.5–3 during peak hours, with each drop lasting 3-5 hours and centered at 8 AM or 6 PM. This empirical way to generate time-dependent costs was not meant to be completely realistic, but at least it should provide "reasonable" data. Then, for each query, we randomly generated with a uniform distribution a departure time in seconds between 7 AM and 7 PM, so that the optimal time-dependent solution has a very high probability of being different than the traffic-free static solution.

To validate the clustered approach we generated a k-center clustering over V, with $k = 100$ clusters, using k'-oversampling with $k' = 200$ (see (103)); that is, we picked 200 random nodes, we connected them to a "dummy" central node, and we grew clusters of neighbouring nodes around each of the 200 centers. Then, when all nodes had been assigned to a cluster, we progressively deleted the smallest remaining cluster, i.e. the one with the smallest radius, allowing other clusters to grow into the deleted one. We iterated this procedure until 100 clusters were left. We compared the number of explored and settled nodes between a Dijkstra search and Algorithm 2, where source and destination node were chosen at random. We also compared the results with respect to the naive algorithm of computing the shortest path in the static traffic-free graph, i.e. G_λ, and then applying time-dependent costs. Results are reported in Table 2.1. For each value of K (first column), we indicate the average number of settled nodes in 1000 Dijkstra searches on the full graph, the average number of settled nodes with Algorithm 2 and the same source-destination pairs, the average percentage increase P of the naive solution value with respect to the optimum (that is, if p^* is the optimal solution and p is the naive solution, the average value of $(1 - \gamma(p, \tau_0)/\gamma(p^*, \tau_0))$), the average percentage increase of the approximated solution value with respect to the optimum, the average CPU time savings of Algorithm 2 in percentage of the CPU times saved with respect to the exact algorithm (0% means as slow as the exact algorithm; a negative value means that there is an increase in CPU time, while a positive value means that CPU time decreased), and the percentage of shortest path computations where the approximated a solution had a cost smaller or equal than the cost of the naive solution. We do not provide exact query times because those are highly dependent on the implementation, and in this case our implementation was not fine-tuned; what is most interesting, here, is the speed-up with respect to plain Dijkstra's algorithm in terms of number of settled nodes and of relative CPU time. The number of settled nodes for the naive approach is not relevant: many speed-up techniques exist for the static case (see Section 1.4), so we can assume that it is a fast computation. While for low values of K computational times could increase, due to the overhead for constraining the Dijkstra search within the boundaries of the guarantee region, for high enough values of the approximation constant the savings in CPU time are significant, with a small average decrease of the solution quality with respect to the optimum. For $K \leq 3$, there is no speedup in the computation. Therefore, this method requires a very large

MAX K	# SETTLED NODES DIJKSTRA	# SETTLED NODES APPROX	SOLUTION COST INCREASE NAIVE	SOLUTION COST INCREASE APPROX	CPU TIME SAVINGS	IMPROVED PATHS
3	185514	60045	4.56%	1.10%	53.99%	91.8%
3.5	194640	35561	4.43%	4.91%	74.66%	74.5%
4	190077	15597	4.58%	9.27%	87.69%	53.2%
4.5	193240	9943	4.46%	16.51%	91.51%	38.2%
3.5*	188988	29341	4.38%	1.75%	78.22%	76.4%
4*	184327	17256	4.79%	4.54%	86.28%	67.2%
4.5*	190944	12675	4.40%	5.40%	91.72%	58.2%

Table 2.1: Computational results on clustered graph: average values. A * in the first column indicates that the value for K has been adaptively chosen, and we report the starting value, which is also the maximum one.

approximation constant to bring practical benefits, although the solution quality is in practice significantly closer to the optimum than the approximation constant would allow. We note, however, that the naive solution has a better average behaviour than our approximated solution for values of $K \geq 3.5$. We tried to investigate the reason behind this. We can see that, for $K = 3.5$, in 74.5% of the shortest path computations the approximated solution is better than the naive one, but in the remaining cases the approximated solution is very far from the optimum, while the naive one isn't. This is due to the fact that, if K is too large, then the guarantee region between two clusters i and j may consist of only the shortest path between c_i and c_j on G_μ. Any approximated solution between those two clusters will pass through that path, which leads to poor performance. For $K = 3$, in 91.8% of the shortest path computations the approximated solution has a cost which is smaller than the cost of the naive solution, so the average behaviour of the approximated solution is satisfying. However, the computation is only 54% faster than a full (unconstrained) Dijkstra search.

To deal with this issue, we adaptively chose the value of the approximation constant K as follows: for each cluster pair, we started with the maximum value for K ($K_{curr} \leftarrow K_{max}$), and if the computed guarantee region included only the path between the cluster centers we decreased the current K by 10% ($K_{curr} \leftarrow 0.9K_{curr}$). We iterated until the guarantee region for that cluster pair had a cardinality which was greater than the number of nodes on the shortest path on G_μ between the cluster centers. Results for this approach are reported in Table 2.1, on the rows with a * in the first column.

With this modification in the guarantee region generation process, we see that the solution quality significantly increases, while still yielding a speed-up in computational time. Moreover, we are able to obtain a better average be-

MAX NODES	# SETTLED NODES DIJKSTRA	# SETTLED NODES APPROX	SOLUTION COST INCREASE NAIVE	SOLUTION COST INCREASE APPROX	CPU TIME SAVINGS	IMPROVED PATHS
50000	189994	23238	3.77%	3.98%	81.91%	74.8%
65000	190698	27159	3.73%	3.13%	78.54%	77.0%
80000	196529	37771	3.72%	2.71%	71.32%	81.4%

Table 2.2: Computational results on clustered graph with a maximum number of settled nodes for each computation: average values.

haviour than the naive approach for larger values of K that allow for an increased speed-up factor, which is a necessary requirement to state that our approach can be useful in practice. We can also see that, if we compare the number of paths where the approximated solution is better than the naive one, there is an improvement with respect to the basic version of the algorithm. Although for $K = 3.5$ the naive solution will be better than the one computed with our algorithm almost 25% of the times, from a practical point of view the approximated solution has much more value with respect to the naive one, because it changes dynamically reflecting traffic changes; on the other hand, the naive solution between two points is always fixed regardless of the time of the day, which is negatively perceived by users.

We also tested the performance of the approach described in Section 2.3 with a maximum number of settled nodes for each point-to-point computation. In order to do this, we initially set $K = 3$, and if necessary we increase its value until all regions comprise a number of nodes smaller than a given threshold. In (2.3), we set $b = 4000$. Results are reported in Table 2.2 (same columns as in previous tables). We can easily observe that the algorithm's performance does not decrease, and if we are willing to settle up to 65000 nodes for each shortest path computation then our method finds a path which is on average better than the naive solution, while still yielding a speed-up factor of almost 5 with respect to plain Dijkstra's algorithm. However, the starting value of K is still too large for industrial applications.

2.4.3 Drawbacks of guarantee regions

The most evident drawback of the algorithm presented in this section is that we must use a very large approximation constant K in order to achieve a significant speedup during the computations. We have seen that in practice the solution quality does not deteriorate as much as K would allow, but from an industrial point of view an approximation guarantee ≥ 3 is not appealing.

There is one additional major drawback, which is difficult to overcome: the space consumption to store guarantee regions is very large. Associating bit flags to each node is very expensive: elementary calculations show that, for a graph

with 18M nodes such as the European road network (Section 5.1), 8GB of memory would allow only ≈ 60 clusters in the graph covering. In this case, each cluster would be very large, hence performance would significantly decrease. More efficient methods for identifying nodes belonging to a guarantee region could be devised, but, as discussed in Section 2.4.1, one has to be careful not to introduce too many false positives. In the end, we decided to try a different approach, in which the set of nodes to be explored is not predetermined and stored in memory, but is computed "on-the-fly" for each shortest path query. This way, there is no additional space consumption for the storage of guarantee regions.

Chapter 3

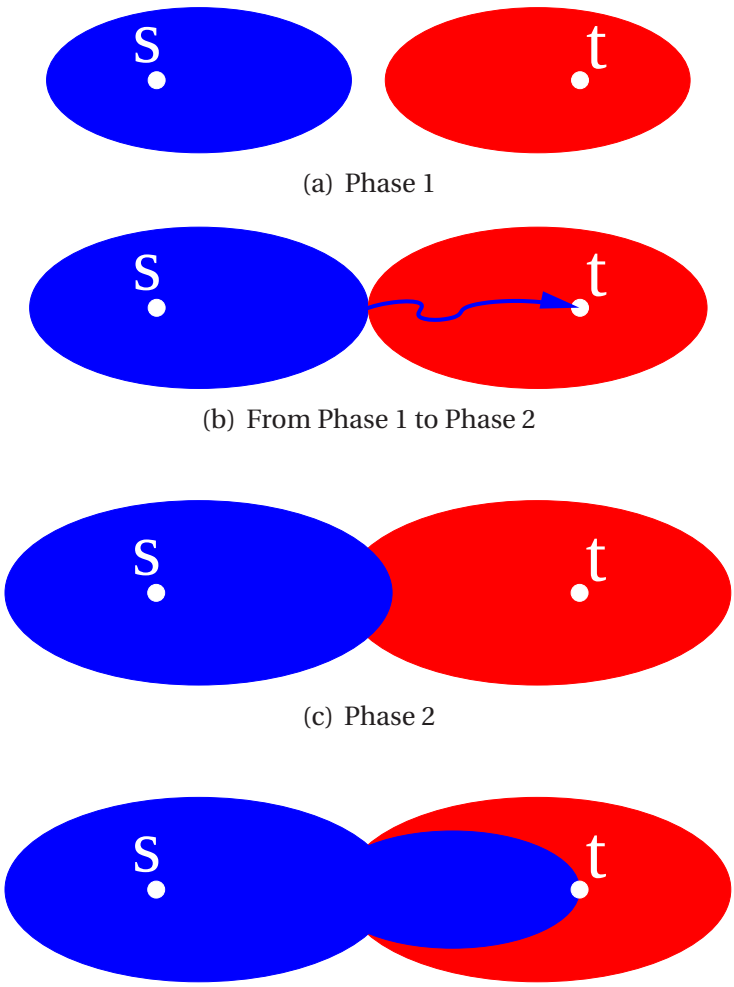
Bidirectional A^* Search on Time-Dependent Graphs

Bidirectional search is an important tool for the development of speedup techniques for the SPP on static graphs; however, it cannot be directly applied on time-dependent graphs. In this section we propose a novel algorithm for the TDSPP based on a bidirectional A^* algorithm. Since the arrival time is not known in advance (so c cannot be evaluated on the arcs adjacent to the destination node), our backward search occurs on the graph weighted by the lower bounding function λ. This is used for bounding the set of nodes that will be explored by the forward search.

The rest of this chapter is organized as follows. In Section 3.1 we describe the foundations of our idea, and present an adaptation of the ALT algorithm based on it. In Section 3.2 we formally prove our method's correctness. In Section 3.3 we propose some modifications that improve the performance of our algorithm, and prove their correctness. In Section 3.4 we discuss the dynamic case, where arc cost functions are allowed to change under weak conditions.

3.1 Algorithm description

We propose a general approach for bidirectional search based on restricting the scope of a time-dependent A^* search from the source using a set of nodes defined by a time-*independent* A^* search from the destination, i.e. the backward search is a reverse search in G_λ. This idea is related to the guarantee regions discussed in Chapter 2: in that case we predetermined the set of nodes that had to be explored for each shortest path computation, while this approach tries to identify such nodes "on-the-fly", depending on the source and destination node. Note that Dijkstra's algorithm is equivalent to A^* with a zero potential function; thus, in the following we could use Dijkstra's algorithm instead of A^*. However, as A^* already yields a reduced search space with respect to Dijkstra's



(a) Phase 1

(b) From Phase 1 to Phase 2

(c) Phase 2

(d) Phase 3

Figure 3.1: Schematic representation of the TDALT algorithm search space

algorithm, we will assume that the first algorithm is used, even though all theoretical properties also hold for Dijkstra's algorithm.

Given a graph $G = (V, A)$ and source and destination vertices $s, t \in V$, the algorithm for computing the shortest time-dependent cost path p^* works in three phases.

1. A bidirectional A^* search occurs on G, where the forward search is run on the graph weighted by c, and the backward search is run on the graph weighted by the lower bounding function λ. All nodes settled by the backward search are included in a set M. Phase 1 terminates as soon as the two search scopes meet.

2. Suppose that $v \in V$ is the first vertex in the intersection of the heaps of the forward and backward search; then the time dependent cost $\mu = \gamma(p_v, \tau_0)$ of the path p_v going from s to t passing through v is an upper bound to $\gamma(p^*, \tau_0)$. In the second phase, both search scopes are allowed to proceed until the backward search queue only contains nodes whose associated key exceeds μ. In other words: let β be the key of the minimum element of the backward search queue; phase 2 terminates as soon as $\beta > \mu$. Again, all nodes settled by the backward search are included in M.

3. Only the forward search continues, with the additional constraint that only nodes in M can be explored. The forward search terminates when t is settled.

If the ALT algorithm (Section 1.4.5.1) is used for the A^* search, i.e. we use landmark potential functions, then we call this algorithm TIME-DEPENDENT ALT (TDALT). The pseudocode is given in Algorithm 3. Note that we use the symbol $\leftrightarrow$ to indicate either the forward search ($\leftrightarrow = \rightarrow$) or the backward search ($\leftrightarrow = \leftarrow$): as a consequence, $\overrightarrow{\pi}$ is the potential function for the forward search, while $\overleftarrow{\pi}$ is the potential function for the backward search. Similarly for all other identifiers. We denote by $\overrightarrow{A}$ the set of arcs for the forward search, i.e. $\overrightarrow{A} = A$, and by $\overleftarrow{A}$ the set of arcs for the backward search, i.e. $\overleftarrow{A} = \overline{A}$. A typical choice is to alternate between the forward and the backward search at each iteration of the algorithm during the first two phases. Through the rest of this work we will always alternate between the two searches at each iteration.

We give a sketch of the algorithm in Figure 3.1. In Figure 3.1(a) the algorithm is in Phase 1: a forward search starts on the time-dependent graph and a backward search starts on the same graph weighted by the lower bounding function λ. As soon as the two search scopes meet (Figure 3.1(b)), we compute an upper bound μ on the cost of optimal solution by evaluating the time-dependent cost of an $s \rightarrow t$ path, and switch to Phase 2. In Figure 3.1(c) the algorithm is in Phase 2: both searches continue until the condition $\beta > \mu$ is verified, where β is the minimum element of the backward search priority queue. When the condition holds, the algorithm switches to Phase 3 (Figure 3.1(d)): the backward search stops, and the forward search continues until the target node is settled, but the forward search is allowed to explore only nodes already settled by the backward search.

3.2 Correctness

We prove correctness of our approach for the computation of both optimal solutions and suboptimal solutions with an approximation guarantee.

Theorem 3.2.1. *Algorithm 3 computes the shortest time-dependent path from s to t for a given departure time τ_0.*

Algorithm 3 TDALT: Compute the shortest time-dependent path from s to t with departure time τ_0

$\overrightarrow{Q}$.insert$(s,0)$; $\overleftarrow{Q}$.insert$(t,0)$; $M := \emptyset$; $\mu := +\infty$; $done :=$ `false`; $phase := 1$.
while $\neg done$ **do**
  if $(phase = 1) \vee (phase = 2)$ **then**
    $\leftrightarrow \in \{\rightarrow, \leftarrow\}$
  else
    $\leftrightarrow := \rightarrow$
  $u := \overleftrightarrow{Q}$.extractMin()
  if $(u = t) \wedge (\leftrightarrow = \rightarrow)$ **then**
    $done :=$ `true`
    continue
  if $(phase = 1) \wedge (u.\text{dist}^{\rightarrow} + u.\text{dist}^{\leftarrow} < \infty)$ **then**
    $\mu := u.\text{dist}^{\rightarrow} + u.\text{dist}^{\leftarrow}$
    $phase := 2$
  if $(phase = 2) \wedge (\leftrightarrow = \leftarrow) \wedge (\mu < u.\text{key}^{\leftarrow})$ **then**
    $phase := 3$
    continue
  for all arcs $(u,v) \in \overleftrightarrow{A}$ **do**
    if $\leftrightarrow = \leftarrow$ **then**
      M.insert(u)
    else if $(phase = 3) \wedge (v \notin M)$ **then**
      continue;
    if $(v \in \overleftrightarrow{Q})$ **then**
      if $u.\text{dist}^{\leftrightarrow} + c(u,v,u.\text{dist}^{\leftrightarrow}) < v.\text{dist}^{\leftrightarrow}$ **then**
        $\overleftrightarrow{Q}$.decreaseKey$(v, u.\text{dist}^{\leftrightarrow} + c(u,v,u.\text{dist}^{\leftrightarrow}) + \overleftrightarrow{\pi}(v))$
    else
      $\overleftrightarrow{Q}$.insert$(v, u.\text{dist}^{\leftrightarrow} + c(u,v,u.\text{dist}^{\leftrightarrow}) + \overleftrightarrow{\pi}(v))$
return $t.\text{dist}^{\rightarrow}$

Proof. The forward search of Algorithm 3 is exactly the same as the unidirectional version of the A^* algorithm during the first 2 phases, and thus it is correct; we have to prove that the restriction applied during phase 3 does not interfere with the correctness of the A^* algorithm.

Let μ be an upper bound on the cost of the shortest path; in particular, this can be the cost $\gamma(p_v, \tau_0)$ of the $s \rightarrow t$ path passing through the first meeting point v of the forward and backward search. Let β be the smallest key of the backward search priority queue at the end of phase 2. Suppose that Algorithm 3 is not correct, i.e. it computes a sub-optimal path. Let p^* be the shortest path from s to t with departure time τ_0, and let u be the first node on p^* which is not explored by the forward search; by phase 3, this implies that $u \notin M$, i.e. u has not been settled by the backward search during the first 2 phases of Algorithm 3. Hence,

we have that $\beta \leq \pi_b(u) + d_\lambda(u,t)$; then we have the chain $\gamma(p^*,\tau_0) \leq \mu < \beta \leq \pi_b(u) + d_\lambda(u,t) \leq d_\lambda(s,u) + d_\lambda(u,t) \leq d(s,u,\tau_0) + d(u,t,\tau_0 + d(s,u,\tau_0)) = \gamma(p^*,\tau_0)$, which is a contradiction. □

Theorem 3.2.2. *Let p^* be the shortest path from s to t. If the condition to switch to phase 3 is $\mu < K\beta$ for a fixed parameter K, then Algorithm 3 computes a path p from s to t such that $\gamma(p,\tau_0) \leq K\gamma(p^*,\tau_0)$ for a given departure time τ_0.*

Proof. Suppose that $\gamma(p,\tau_0) > K\gamma(p^*,\tau_0)$. Let u be the first node on p^* which is not explored by the forward search; by phase 3, this implies that $u \notin M$, i.e. u has not been settled by the backward search during the first 2 phases of Algorithm 3. Hence, we have that $\beta \leq \pi_b(u) + d_\lambda(u,t)$; then we have the chain $\gamma(p,\tau_0) \leq \mu < K\beta \leq K(\pi_b(u) + d_\lambda(u,t)) \leq K(d_\lambda(s,u) + d_\lambda(u,t)) \leq K(d(s,u,\tau_0) + d(u,t,\tau_0 + d(s,u,\tau_0))) = K\gamma(p^*,\tau_0) < \gamma(p,\tau_0)$, which is a contradiction. □

3.3 Improvements

Performance of the basic version of the algorithm can be improved with the results that we describe in this section, whose purpose is to reduce further the size of the search space.

Theorem 3.3.1. *Let p be the shortest path from s to t with departure time τ_0. If all nodes u on p settled by the backward search are settled with a key smaller or equal to $d(s,u,\tau_0) + d(u,t,\tau_0 + d(s,u,\tau_0))$, then Algorithm 3 is correct.*

Proof. Let Q be the backward search queue, let key(u) be the key for the backward search of node u, let $\beta = \min_{u \in Q}\{\text{key}(u)\}$ be the smallest key in the backward search queue, which is attained at a node v (i.e. $v = \arg\min_{u \in Q}\{\text{key}(u)\}$), and let μ the best upper bound on the cost of the solution currently known. To prove correctness, using the same arguments as in the proof of Theorem 3.2.1 we must make sure that, when the backward search stops at the end of phase 2, then all nodes on the shortest path from s to t that have not been explored by the forward search have been added to M. The backward search stops when $\mu < \beta$.

In an A^* search, the keys of settled nodes are non-decreasing. So every node u which at the current iteration has not been settled by the backward search will be settled with a key key$(u) \geq$ key(v), which yields $d(s,u,\tau_0) + d(u,t,\tau_0 + d(s,u,\tau_0)) \geq \text{key}(v) = \beta > \mu \ \forall u \in Q$. Thus, every node which has not been settled by the backward search cannot be on the shortest path from s to t, and Algorithm 3 is correct. □

This allows the use of larger lower bounds during the backward search: the backward A^* search does not have to compute shortest paths on the graph G_λ, but it should in any case guarantee that when a node u is settled then its key is an underestimation of the time-dependent cost of the time-dependent shortest path between s and t passing through u. The next proposition is of fundamental practical importance.

Proposition 3.3.2. *In phase 2 of Algorithm 3, nodes that have already been settled by the forward search do not have to be explored by the backward search.*

Proof. Let $d_b(v)$ be the distance from a node v to node t computed by the backward search if we do not explore any node already explored by the forward search. Obviously we have $d_b(v) \geq d_\lambda(v,t)$. We will prove that, when a node v on the shortest path from s to t with departure time τ_0 is settled by the backward search, then $d_b(v) \leq d(v,t,\tau_0 + d(s,v,\tau_0)) \; \forall \tau_0 \in \mathcal{T}$. By Theorem 3.3.1, this is enough to prove our statement.

Consider a node v settled by the backward search, but not by the forward search; let q be the shortest path from s to v with departure time τ_0, let q' be the shortest path from v to t with departure time $tau_0 + \gamma(q,\tau_0)$. Suppose that q' does not pass through any node already settled by the forward search. Then $d_b(v) \leq \lambda(q') \leq d(v,t,\tau_0 + d(s,v,\tau_0))$.

Suppose now that q' passes through a node w already settled by the forward search. Let p be the shortest path from s to w with departure time τ_0, and let p' be the shortest path from w to t with departure time $\tau_0 + \gamma(p,\tau_0)$; clearly v cannot be on p, because otherwise it would have been settled by the forward search. By optimality of p we have $\gamma(p,\tau_0) \leq \gamma(q + q'_{|v\to w},\tau_0) = \gamma(q,\tau_0) + \gamma(q'_{|v\to w},\tau_0 + \gamma(q,\tau_0))$. Thus, by the FIFO property, we have the chain $\gamma(p+p',\tau_0) = \gamma(p,\tau_0) + \gamma(p',\tau_0 + \gamma(p,\tau_0)) \leq \gamma(q,\tau_0) + \gamma(q'_{|v\to w},\tau_0 + \gamma(q,\tau_0)) + \gamma(q'_{|w\to t},\tau_0 + \gamma(q,\tau_0) + \gamma(q'_{|v\to w},\tau_0 + \gamma(q,\tau_0))) = \gamma(q+q',\tau_0)$, which means that v does not have to be explored and added to the set M by the backward search, because we already have a better path passing through w. Thus, even if $\text{key}(v) > d(s,v,\tau_0) + d(v,t,\tau_0 + d(s,v,\tau_0))$ Algorithm 3 is correct. □

By Theorem 3.3.1, we can take advantage of the fact that the backward search is used only to bound the set of nodes explored by the forward search. This means that we can tighten the bounds used by the backward search, even if doing so would result in an A^* backward search that computes suboptimal distances. To derive some valid lower bounds we need the following lemma and propositions.

Lemma 3.3.3. *Let v be a node, and u its parent node in the shortest path from s to v with departure time τ_0. Then $d(s,u,\tau_0) + \pi_f(u) \leq d(s,v,\tau_0) + \pi_f(v)$.*

Proof. Suppose that ℓ is the active landmark, i.e. the landmark in our landmarks set that currently gives the best bound; we have that either $\pi_f(u) = d_\lambda(u,\ell) - d_\lambda(t,\ell)$ or $\pi_f(u) = d_\lambda(\ell,t) - d_\lambda(\ell,u)$.

First case: $\pi_f(u) = d_\lambda(u,\ell) - d_\lambda(t,\ell)$. We have $d(s,u,\tau_0) + \pi_f(u) = d(s,u,\tau_0) + d_\lambda(u,\ell) - d_\lambda(t,\ell) \leq d(s,u,\tau_0) + d_\lambda(u,v) + d_\lambda(v,\ell) - d_\lambda(t,\ell) \leq d(s,u,\tau_0) + \lambda(u,v) + d_\lambda(v,\ell) - d_\lambda(t,\ell) \leq d(s,v,\tau_0) + \pi_f(v)$.

Second case: $\pi_f(u) = d_\lambda(\ell,t) - d_\lambda(\ell,u)$. We have $d(s,u,\tau_0) + \pi_f(u) = d(s,u,\tau_0) + d_\lambda(\ell,t) - d_\lambda(\ell,u)$; by triangular distance, $d_\lambda(\ell,v) \leq d_\lambda(\ell,u) + d_\lambda(u,v) \leq d_\lambda(\ell,u) + \lambda(u,v)$, which yields $-d_\lambda(\ell,u) \leq -d_\lambda(\ell,v) + \lambda(u,v)$. So $d(s,u,\tau_0) + d_\lambda(\ell,t) - d_\lambda(\ell,u) \leq d(s,u,\tau_0) + d_\lambda(\ell,t) - d_\lambda(\ell,v) + \lambda(u,v) \leq d(s,v,\tau_0) + \pi_f(v)$. □

Proposition 3.3.4. *At a given iteration, let v be the last node settled by the forward search. Then, for each node w which has not been settled by the forward search,* $d(s,v,\tau_0) + \pi_f(v) - \pi_f(w) \leq d(s,w,\tau_0)$.

Proof. There are two possibilities for w: either it has been explored (but not settled) by the forward search, or it has not been explored. Let Q be the set of nodes in the forward search queue. If w has been explored, then $w \in Q$, and clearly $d(s,v,\tau_0) + \pi_f(v) \leq d(s,w,\tau_0) + \pi_f(w)$ because v has been extracted before w, which proves our statement. Otherwise, there is a node $u \in Q$ on the shortest path from s to w with departure time τ_0 which has been explored but not settled. We have that $d(s,v,\tau_0) + \pi_f(v) \leq d(s,u,\tau_0) + \pi_f(u)$ because v has been extracted while u is still in the queue, and by Lemma 3.3.3, if we examine the nodes $u = u_1, u_2, \ldots, u_k = w$ on the shortest path from s to w with departure time τ_0, we have that $d(s,u_1,\tau_0) + \pi_f(u_1) \leq \cdots \leq d(s,u_k,\tau_0) + \pi_f(u_k)$, from which our statement follows. □

Let v be as in Proposition 3.3.4, and w a node which has not been settled by the forward search. Proposition 3.3.4 suggests that we can use

$$\pi_b^*(w) = \max\{\pi_b(w), d(s,v,\tau_0) + \pi_f(v) - \pi_f(w)\} \tag{3.1}$$

as a lower bound to $d(s,w,\tau_0)$ during the backward search. However, we have to make sure that the bound is valid at each iteration of Algorithm 3.

Lemma 3.3.5. *If the key of the forward search used to compute the potential function π_b^* defined by (3.1) is fixed, then we have $\pi_b^*(v) \leq \pi_b^*(u) + \lambda(u,v)$ for each arc $(u,v) \in A$.*

Proof. By definition we have $\pi_b^*(v) = \max\{\pi_b(v), \alpha - \pi_f(v)\}$, where with α we denoted the key of a node settled by the forward search, which is fixed by hypothesis. Consider the case $\pi_b^*(v) = \pi_b(v)$; then, since the landmark potential functions π_b and π_f are consistent, we have $\pi_b^*(v) = \pi_b(v) \leq \pi_b(u) + \lambda(u,v) \leq \pi_b^*(u) + \lambda(u,v)$. Now consider the case $\pi_b^* = \alpha - \pi_f(v)$; then we have $\pi_b^*(v) = \alpha - \pi_f(v) \leq \alpha - \pi_f(u) + \lambda(u,v) \leq \pi_b^*(u) + \lambda(u,v)$, which completes the proof. □

This is enough to prove correctness of our algorithm with tightened bounds, as stated in the next theorem.

Theorem 3.3.6. *If we use the potential function π_b^* defined by (3.1) as potential function for the backward search, with a fixed value of the forward search key, then Algorithm 3 is correct.*

Proof. Let $d_b(u)$ be the distance from a node u to node t computed by the backward search. We will prove that, when a node u on the shortest path from s to t is settled by the backward search, $d_b(u) \leq d(u,t,\tau_0 + d(s,u,\tau_0)) \ \forall \tau_0 \in T$. By Proposition 3.3.4 and Theorem 3.3.1, this is enough to prove our statement.

Let $q^* = (v_1 = u, \ldots, v_n = t)$ be the shortest path from u to t on G_λ. We proceed by induction on $i : n, \ldots, 1$ to prove that each node v_i is settled with the correct distance on G_λ, i.e. $d_b(v_i) = d_\lambda(v_i, t)$. It is trivial to see that the nodes v_n and v_{n-1} are settled with the correct distance on G_λ. For the induction step, suppose v_i is settled with the correct distance $d_b(v_i) = d_\lambda(v_i, t)$. By Lemma 3.3.5, we have $d_b(v_i) + \pi_b^*(v_i) \leq d_b(v_i) + \lambda(v_{i-1}, v_i) + \pi_b^*(v_{i-1}) = d_\lambda(v_{i-1}, t) + \pi_b^*(v_{i-1}) \leq d_b(v_{i-1}) + \pi_b^*(v_{i-1})$, hence v_i is extracted from the queue before v_{i-1}. This means that v_{i-1} will be settled with the correct distance $d_b(v_{i-1}) = d_\lambda(v_{i-1}, t)$, and the induction step is proven.

Thus, u will be settled with distance $d_b(u) = d_\lambda(u,t) \leq d(u,t,\tau_0 + d(s,u,\tau_0))$, which proves our statement. □

By Theorem 3.3.6, Algorithm 3 is correct when using π_b^* only if we assume that the node v used in (3.1) is fixed at each backward search iteration. Thus, in practice we do the following: we set up $k_{\max} \in \mathbb{N}$ checkpoints during the query; when a checkpoint is reached, the node v used to compute (3.1) is updated, and the backward search queue is flushed and filled again using the updated π_b^*. This is enough to guarantee correctness. The checkpoints are computed comparing the initial lower bound $\Delta = \pi_f(t)$ and the current distance from the source node, both for the forward search: the initial lower bound is divided by $k_{\max}$ and, whenever the current distance from the source node exceeds $k\Delta/k_{\max}$ with $k \in \{1, \ldots, k_{\max}\}$, π_b^* is updated.

3.4 Dynamic cost updates

Up to now, time-dependent routing algorithms assumed complete knowledge of the time-dependent cost functions on arcs. However, since the speed profiles on which these functions are based are generated using historical data gathered from sensors (or cams), it is reasonable to assume that also real-time traffic information is available through these sensors. Moreover, other technologies exist to be aware of traffic jams even without having access to real-time speed information (e.g., TMC[1]). In the end, a procedure to update the time-dependent cost functions depending on real-time traffic information would be desirable for practical applications.

[1] `http://www.tmcforum.com/`

The input required by the TDALT algorithm (Section 3.1) consists of the graph G with the associated arc cost functions and the distances to and from landmarks. As landmark distances are computed using the lower bounding function λ, it is clear that the preprocessing information, i.e. landmark distances, remain valid as long as $\lambda = \underline{c}$, that is, λ bounds c from below. On road networks, computing a lower bounding function which is necessarily valid can be easily done: it suffices to divide the length of the road segment that an arc represents by the maximum speed allowed on that type of road. Thus, as long as $\lambda = \underline{c}$, the time-dependent cost functions can be updated with no additional required effort.

Chapter 4

Core Routing on Time-Dependent Graphs

Hierarchical speedup techniques have been successfully used for the SPP on static graphs (Section 1.4.4), hence in this section we generalize these techniques to the time-dependent scenario and analyse performance of a two-level hierarchical setup (*core routing*) for the TDSPP. The idea behind core routing is to shrink the original graph in order to get a new graph (*core*) with a smaller number of vertices. Most of the search is then carried out on the core, yielding a reduced search space. We combine core routing with the bidirectional goal-directed algorithm TDALT.

The rest of this section is organized as follow. In Section 4.1 we describe core routing on static graphs and generalize it to the time-dependent case. In Section 4.2 we discuss several issues that arise during the practical implementation of a core-based algorithm, and propose solutions. In Section 4.3 we analyse under which conditions core optimality can be rapidly restored in the case of updates in the cost functions, and propose an algorithm for this task. Finally, in Section 4.4 we discuss the extension of a two-levels approach to a multilevel hierarchy.

4.1 Algorithm description

Core-based routing is a powerful approach which has been widely and successfully used for shortest paths algorithms on static graphs (Section 1.4.4. The main idea is to use contraction (Section 1.4.4.3): a routine iteratively removes unimportant nodes and adds edges to preserve correct distances between the remaining nodes, so that we have a smaller network where most of the search can be carried out. Note that in principle we can use *any* contraction routine which removes nodes from the graph and inserts edges to preserve distances. When the contracted graph $G_C = (V_C, A_C)$ has been computed, it is merged

with the original graph to obtain $G_F = G_C \cup G = (V, A \cup A_C)$ since $V_C \subset V$.

Suppose that we have a contraction routine which works on a time-dependent graph: that is, $\forall u, v \in V_C$, for each departure time $\tau_0 \in \mathcal{T}$ there is a shortest path between u and v in G_C with the same cost as the shortest path between u and v in G with the same departure time. We propose the following query algorithm.

1. Initialization phase: start a Dijkstra search from both the source and the destination node on G_F, using the time-dependent costs for the forward search and the time-independent costs λ for the backward search, pruning the search (i.e. not relaxing outgoing arcs) at nodes $\in V_C$. Add each node settled by the forward search to a set S, and each node settled by the backward search to a set T. Iterate between the two searches until: (i) $S \cap T \neq \emptyset$ or (ii) the priority queues are empty.

2. Main phase:

   - (i) If $S \cap T \neq \emptyset$, then start an unidirectional Dijkstra search from the source on G_F until the target is settled.
   - (ii) If the priority queues are empty and we still have $S \cap T = \emptyset$, then start TDALT on the graph G_C, initializing the forward search queue with all leaves of S and the backward search queue with all leaves of T, using the distance labels computed during the initialization phase. The forward search is also allowed to explore any node $v \in T$, throughout the 3 phases of the algorithm. Stop when t is settled by the foward search.

In other words, the forward search "hops on" the core when it reaches a node $u \in S \cap V_C$, and "hops off" at all nodes $v \in T \cap V_C$. Again, since Dijkstra's algorithm is equivalent to A^* with a zero potential function, we can use Dijkstra's algorithm in case (ii) during the main phase. It is interesting to note that, independently from our work, this approach to apply hierarchical speed up techniques to time-dependent road networks has also been proposed in (17), in an effort to generalize the Contraction Hierarchies algorithm (Section 1.4.4.3) to the time-dependent case. However, no computational results are given. Next, we prove that this core routing approach is correct.

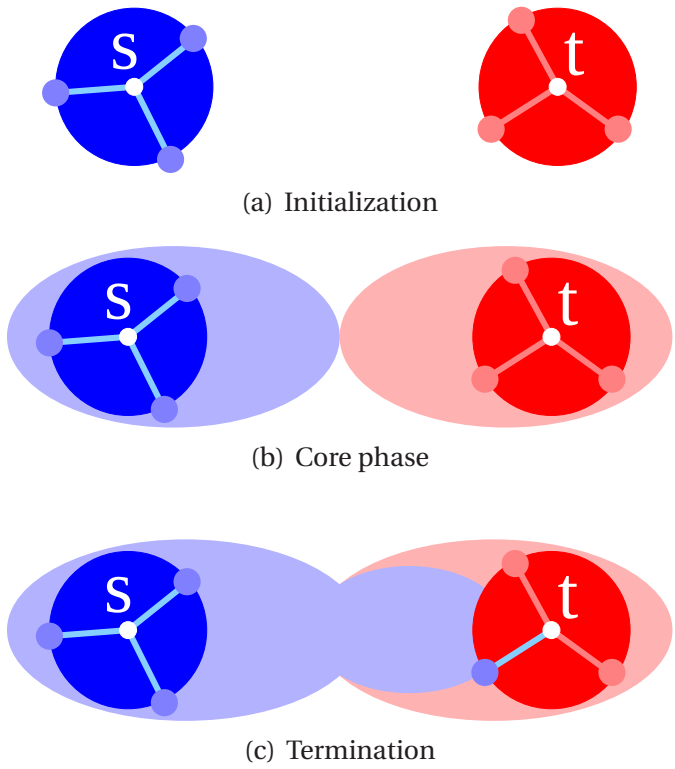
Proposition 4.1.1. *The core routing algorithm for time-dependent graphs is correct.*

Proof. Suppose that, during the initialization phase (i.e. when we build the two sets S and T), the two search scopes meet, thus $S \cap T \neq \emptyset$. In this case, we switch to unidirectional Dijkstra's algorithm on the original graph (plus added shortcuts), and correctness follows. Now suppose that the two search scopes do not meet: the two priority queues are empty and $S \cap T = \emptyset$, thus the shortest path p

between s and t with departure time τ_0 passes through at least one node belonging to the core V_C. Let $p = (s, \ldots, u, \ldots, v, \ldots, t)$, where u and v are, respectively, the first and the last node $\in V_C$ on the path. If $u = v$ then the proof is trivial; suppose $u \neq v$. Since the initialization phase explores all non-core nodes reachable from s and t, $u \in S$ and $v \in T$. By definition of v, $p_{|v \to t}$ passes only through non-core nodes; by the query algorithm, T contains all non-core nodes that can reach t passing only through non-core nodes. It follows that all nodes of $p_{|v \to t}$ are in T. Thus $p_{|u \to t}$ is entirely contained in $G_C \cup G[T] = (V_C \cup T, A_C \cup A[T])$. By correctness of Dijkstra's algorithm, the distance labels for nodes in S are exact with respect to the time-dependent cost function. Initializing the forward search queue with the leaves of S and applying A^* on $G_C \cup G[T]$ then yields the shortest path p by correctness of A^*. □

We immediately observe that for case (ii) of the main phase we can use any algorithm that guarantees correctness when applied on $G_C \cup G(T)$. In particular, the distance labels for nodes in T are correct distance labels for the backward search on the graph weighted by λ, so they fulfill the requirements for TDALT (Section 3.1). Note that, in a typical core-routing setting for the ALT algorithm, landmark distances are computed and stored only for vertices in V_C (see (19)), since the initialization phase on non-core nodes uses Dijkstra's algorithm only. This means that the landmark potential function cannot be used to apply the forward A^* search on the nodes in T. However, in order to combine TDALT with a core-routing framework we can use the backward distance labels computed with Dijkstra's algorithm during the initialization phase. Those are correct distance labels for the lower bounding function λ, thus they yield valid potentials for the forward search. We call this algorithm TIME-DEPENDENT CORE-BASED ALT (TDCALT).

We give a skeatch of the TDCALT algorithm search space in Figure 4.1. In Figure 4.1(a), the algorithm is initialized: a forward search is started from the source using time-dependent costs and a backward search is started from the target using the static costs λ, but arcs in the outstar of core nodes are *not* relaxed. The leaves of the shortest paths trees rooted at the source and the destination are the access points to the core. If the two search spaces do not intersect during the initialization phase, then the TDALT algorithm restricted to the core is applied (Figure 4.1(b)), using the previously computed access points to initialize the forward and backward search queue. As the core comprises a significantly smaller number of nodes and arcs with respect to the original space, the search space shrinks by a large factor. The forward search is then allowed to explore non-core nodes reachable from the access points near the destination (Figure 4.1(c)), until the target is settled and the algorithm terminates.



(a) Initialization

(b) Core phase

(c) Termination

Figure 4.1: Schematic representation of the TDCALT algorithm search space

4.2 Practical issues

There are still several missing pieces before a full description of a practical implementation of the algorithm described in Section 4.1 can be given. Namely, we should describe a way to compute the potential function for nodes in $V \setminus (V_C \cup T)$, discuss the contraction routing, and give an algorithm to retrieve the full shortest path on the original graph when the computations are done on the contracted graph.

4.2.1 Proxy nodes

Since landmark distances are available only for nodes in V_C, the ALT potential function cannot be used "as is" whenever the source or the destination node do not belong to the core. In order to compute valid lower bounds to the distances from s or to t, proxy nodes have been introduced in (69) and used for the CALT algorithm (i.e. core-based ALT on a static graph) in (19). See also (71) for a description. We report here the main idea: on the graph G weighted by λ, let $t' = \arg\min_{v \in V_C}\{d(t,v)\}$ be the core node closest to t. By

triangle inequalities it is easy to derive a valid potential function for the forward search which uses landmark distances for t' as a proxy for t: $\pi_f(u) = \max_{\ell \in L}\{d(u,\ell) - d(t',\ell) - d(t,t'), d(\ell,t') - d(\ell,u) - d(t,t')\}$. The same calculations yield the potential function for the backward search π_b using a proxy node s' for the source s and the distance $d(s',s)$.

4.2.2 Contraction

For the contraction phase, i.e. the routine which selects which nodes have to be bypassed and then adds shortcuts to preserve shortest paths, we use the algorithm proposed in (41). We define the *expansion* (69) of a node u as the quotient between the number of added shortcuts and the number of edges removed if u is bypassed, and the *hop-number* of a shortcut as the number of edges that the shortcut represents. We iterate the contraction routine until the expansion of all remaining nodes exceeds a limit C or the hop-number exceeds a limit H. Note that, in the case of piecewise linear cost functions, the composition of two functions yields an increase in the number of breakpoints; indeed, if two functions $f, g \in \mathbb{F}$ have, respectively, $B(f)$ and $B(g)$ breakpoints, then the composition $f \oplus g$ may have up to $B(f) + B(g)$ breakpoints in the worst case. As these points have to be stored in memory, the space consumption may become unpractical if we add too many long shortcuts. Thus, we also enforce a limit I on the maximum number of breakpoints that each shortcut may have: if a shortcut which exceeds this limit would be created, we simply skip it.

In order to choose which node has to be bypassed at each step of the contraction routine, we mantain a heap of all nodes sorted by a function value (*bypassability score*) which favours nodes whose bypassing creates fewer and shorter shortcuts, and extract the minimum element at each iteration. The bypassability score of a node u is defined as a linear combination of: the expansion of u, the hop-number of the longest shortcut that would be created if u is bypassed, and the largest number of interpolation points of the shortcuts that would be created if u is bypassed. Note that the contraction of u may influence the bypassability score of adjacent nodes, so these scores must be recomputed after a node is chosen. As suggested by (41), we give a larger importance to the expansion of a node when determining its bypassability score, thus using a coefficient of 10 for this factor in the linear combination, whereas the other two factors are added with unitary coefficient. At the end of the contraction routine, we perform an edge-reduction step which removes unnecessary shortcuts from the graph. In particular, for each node of the core $u \in V_C$ we check whether for each arc $(u,v) \in A_C$ there is a path p from u to v which does not use the arc (u,v) and such that $\gamma_\tau(p) < c(u,v,\tau)\ \forall \tau \in \mathcal{T}$. This step can be performed by computing the cost function $d_*(u,v)$ on the graph $(V_C, A_C \setminus \{(u,v)\})$ with a label-correcting algorithm (Section 1.4.3) and comparing $d_*(u,v)$ with $c(u,v)$. If $d_*(u,v)(\tau) < c(u,v,\tau)\ \forall \tau \in \mathcal{T}$, then the arc (u,v) is not necessary, as

there is a shorter path between u and v for all possible departure times (see also (41)). Whenever a shortcut between two nodes $u, v \in V$ is added, its cost for each time instant of the time interval is computed running a label-correcting algorithm between u and v (Section 1.4.3).

4.2.3 Outputting shortest paths

Shortcuts are added to the graph in order to accelerate queries. However, as for all shortcut-based speedup techniques, those shortcuts have to be expanded if we want to retrieve the complete shortest path and not only the distance. Our contraction routine for time-dependent graphs is an augmented version of the one introduced for Highway Hierarchies (124). In (45), an efficient unpacking routine based on storing all the arcs a shortcut represents is introduced: since arc identifiers may be several bytes long, for each arc (u, v) on the path that the shortcut represents we store the difference between its index and the index of the first outgoing arc of u. As the outstar of each node is stored contiguously in memory for obvious spatial locality reasons, and the outdegree of nodes is tipically small, this difference can be represented in a small number of bits. However, in the static case a shortcut represents exactly one path because between any two nodes we only need to keep track of the *shortest* arc that connects them, whereas in the time-dependent case the shortest arc between two nodes may be different for each different traversal time. We solve this problem by allowing multi-edges: whenever a node is bypassed, a shortcut is inserted to represent each pair of incoming and outgoing edges, even if another edge between the two endpoints already exists. Thus, multiple shortcuts between the same endpoints are not merged. With this modification each shortcut represents exactly one path, so we can directly apply the unpacking routine from (45). In our experimental evaluation the additional computational time to output a full representation of the shortest path is ≈ 1 millisecond (see Chapter 5).

4.3 Dynamic cost updates

Modifications in the cost functions can be easily taken into account under weak assumptions if shortcuts have not been added to the graph (see Section 3.4). However, a two-levels hierarchical setup is significantly more difficult to deal with, exactly because of shortcuts: since a shortcut represents the shortest path between its two endpoints for at least one departure time, if some arc costs change then the shortest path which is represented may also be subject to changes. Thus, a procedure to restore optimality of the core is needed. We first analyse the general case for modifications in the breakpoint values; then we focus on the simpler case of increasing breakpoint values, and finally propose an algo-

rithmic framework to deal with general cost changes under some restrictive assumptions which are acceptable in practice.

4.3.1 Analysis of the general case

Let (V_C, A_C) be the core of G. Suppose that the cost function of one arc $a \in A$ is modified; the set of core nodes V_C need not change, as long as A_C is updated in order to preserve distances with respect to the uncontracted graph $G = (V, A)$ with the new cost function. There are two possible cases: either the new values of the modified breakpoints are smaller than the previous ones, or they are larger. In the first case, all arcs on the core A_C must be recomputed by running a label-correcting algorithm between the endpoints of each shortcut, as we do not know which shortcuts the updated arc may contribute to. This requires a significant computational effort, and should be avoided if we want to perform fast updates for real-time applications. In the second case, the cost function for core arcs may change for all those arcs $a' \in A_C$ such that a' contains a in its decomposition for at least one time instant τ. In other words, if a contributed to a shortcut a', then the cost of a' has to be recomputed. As the cost of a has increased, then a cannot possibly contribute to other arcs, thus we can restrict the update only to the shortcuts that contain the arc. We now analyse this case in further detail.

4.3.2 Increases in breakpoint values

To perform fast updates in the case that breakpoint values increase, we store for each $a \in A$ the set $S(a)$ of all shortcuts that a contributes to. Then, if one or more breakpoints of a have their value changed, we do the following.

Let $[\tau_1, \tau_{n-1}]$ be the smallest time interval that contains all modified breakpoints of arc a. If the breakpoints preceding and following $[\tau_1, \tau_{n-1}]$ are, respectively, at times τ_0 and τ_n the cost function of a changes only in the interval $[\tau_0, \tau_n]$. For each shortcut $a' \in S(a)$, let $a'_0, \ldots, a'_d$, with $a'_i \in A \, \forall i$, be its decomposition in terms of the original arcs, let $\lambda_j = \sum_{i=0}^{j-1} \lambda(a'_i)$ and $\rho_j = \sum_{i=0}^{j-1} \rho(a'_i)$. If a is the arc with index j in the decomposition of a', then a' may be affected by the change in the cost function of a only if the departure time from the starting point of a' is in the interval $[\tau_0 - \rho_j, \tau_n - \lambda_j]$. This is because a can be reached from the starting node of a' no sooner than λ_j, and no later than ρ_j. Thus, in order to update the shortcut a', we need to run a label-correcting algorithm between its two endpoints only in the time interval $[\tau_0 - \rho_j, \tau_n - \lambda_j]$, as the rest of the cost function is not affected by the change. In practice, if the length of the time interval $[\tau_0, \tau_n]$ is larger than a given threshold we run a label-correcting algorithm between the shortcut's endpoints over the whole time period, as the gain obtained by running the algorithm over a smaller time interval does not offset the overhead due to updating only a part of the profile with respect to computing from scratch.

4.3.3 A realistic scenario

The procedure described in Section 4.3.2 is valid only when the value of breakpoints increases. In a typical realistic scenario, this is often the case: the initial cost profiles are used to model normal traffic conditions, and cost updates occur only to add temporary slowdowns due to unexpected traffic jams. When the temporary slowdowns are no longer valid we would like to restore the initial cost profiles, i.e. lower breakpoints to their initial values, without recomputing the whole core. If we want to allow fast updates as long as the new breakpoint values are larger than the ones used for the initial core construction, without requiring that the values can only increase, then we have to manage the sets $S(a)\,\forall a \in A$ accordingly. We provide an example that shows how problems could arise.

Example 4.3.1. *Given $a \in A$, suppose that the cost of its breakpoint at time $\tau \in \mathcal{T}$ increases, and all shortcuts $\in S(a)$ are updated. Suppose that, for a shortcut $a' \in S(a)$, a does not contibute to a' anymore due to the increased breakpoint value. If a' is removed from $S(a)$ and at a later time the value of the breakpoint at τ is restored to the original value, then a' would not be updated because $a' \notin S(a)$, thus a' would not be optimal.*

Our approach to tackle this problem is the following: for each arc $a \in A$, we update the sets $S(a)$ whenever a breakpoint value changes, with the additional constraint that elements of $S(a)$ after the initial core construction phase cannot be removed from the set. Thus, $S(a)$ contains all shortcuts that a contributes to with the current cost function, plus all shortcuts that a contributed to during the initial core construction. As a consequence we may update a shortcut $a' \in S(a)$ unnecessarily, if a contributed to a' during the initial core construction but ceased contributing after an update step; however, this guarantees correctness for all changes in the breakpoint values, as long as the new values are not strictly smaller than the values used during the initial graph contraction. From a practical point of view, this is a reasonable assumption.

Since the sets $S(a)\,\forall a \in A$ are stored in memory, the computational time required by the core update is largely dominated by the time required to run the label-correcting algorithm between the endpoints of shortcuts. Thus, we have a trade-off between query speed and update speed: if we allow the contraction routine to build long shortcuts (in terms of number of bypassed nodes, i.e. "hops", as well as travelling time) then we obtain a faster query algorithm, because we are able to skip more nodes during the shortest path computations. On the other hand, if we allow only limited-length shortcuts, then the query search space is larger, but the core update is significantly faster as the label-correcting algorithm takes less time. In Chapter 5 we provide an experimental evaluation for different scenarios.

4.4 Multilevel Hierarchy

In principle, the hierarchical query algorithm described in Section 4.1 could be generalized to a multilevel hierarchy. The idea is the following: for each level except the topmost one, we apply Dijkstra's algorithm, using the time-dependent costs c for the forward search, and the static costs λ for the backward search. At each level, we do not relax outgoing arcs for nodes that belong to a higher level in the hierarchy, and, as soon as both priority queues are empty, we use all leaves of the Dijkstra search trees as access points to the upper level. Goal directed search (i.e. the TDALT algorithm) is applied only to the topmost level L, assuming that the forward and backward search scopes do not meet before reaching the topmost level. If this is the case, let $l < L$ be the level at which the two search scopes meet; then there are two possibilities: either we switch to a plain Dijkstra search on level l, or we apply TDALT on level l. The latter option has the drawback of requiring landmark distances to be available for level l, while typically they are computed only for the topmost level L to save preprocessing time and space.

We can formalize this as follows. Let $G_C^l = (V_C^l, A_C^l)$ for $l = 0, \ldots, L$ be a hierarchy of graphs such that $V_C^l \subset V_C^{l+1}$ $\forall l = 0, \ldots, L-1$ and $G_C^0 = G$. Again, we assume that $\forall l = 0, \ldots, L, \forall u, v \in V_C^l$, for each departure time $\tau_0 \in \mathcal{T}$ there is a shortest path between u and v in G_C^l with the same cost as the shortest path between u and v in G with the same departure time. A path can be computed with the following algorithm.

1. Initialization: set $l = 0, S = \{s\}, T = \{t\}$.

2. Level selection phase: start a Dijkstra search from both the source and the destination node on G_C^l, initializing the forward search queue with all leaves of S and the backward search queue with all leaves of T, using the time-dependent costs for the forward search and the time-independent costs λ for the backward search. The search must be pruned (i.e. outgoing arcs should not be relaxed) at nodes $\in V_C^{l+1}$. Add each node settled by the forward search to a set S, and each node settled by the backward search to a set T. Iterate between the two searches until: (i) $S \cap T \neq \emptyset$ or (ii) the priority queues are empty.

3. Main phase:

   - (i) If $S \cap T \neq \emptyset$, then start an unidirectional Dijkstra search from the source on G_F until the target is settled.
   - (ii) If the priority queues are empty with $S \cap T = \emptyset$, then if $l < L$, set $l = l + 1$ and return to 2. Otherwise, start TDALT on the graph G_C^L, initializing the forward search queue with all leaves of S and the backward search queue with all leaves of T, using the distance labels

computed during the initialization phase. The forward search is also allowed to explore any node $v \in T$, throughout the 3 phases of the algorithm. Stop when t is settled by the foward search.

Note that, at each iteration of step 2 (level selection phase), more nodes are added to the two sets S and T are modified, or, in other words, the two shortest paths trees associated with S and T grow; therefore, the leaves of the two sets are different at each iteration. The correctness proof is almost identical to the one for Proposition 4.1.1, hence we omit it.

In static graphs, multilevel hierarchical methods have shown very good results in practice(see Section 1.4.4). However, this does not seem to be true for the time-dependent case. Computational experiments (Chapter 5) indicate that the complexity of shortcuts grows rapidly if we apply a strong contraction to the original graph. Even for a two-levels hierarchical setup, space consumption for the interpolation points of shortcuts may easily become unpractical. Moreover, our experiments have shown that the dynamic scenario (i.e. updates in the time-dependent cost functions) can be dealt with efficiently only if the shortcuts are not too long, both in terms of number of original arcs that they represent and of travel time weighted by λ. A multilevel hierarchical approach further increases the length of shortcuts, as the contraction of each level above level 0 has to combine several of them together, instead of simply combining original arcs. Besides, it is not clear whether more levels in the hierarchy would bring an advantage in terms of reduced query times, as having more levels increases the possibility that the forward and backward search scopes meet before reaching the topmost level, in which case the algorithm's behaviour is not optimal. (17) describes a multilevel approach, but does not provide computational experiments to show its feasibility in practice. For all these reasons, it does not seem a good idea in practice to use a multilevel setup, thus we decided to test only a two-levels hierarchy in the following.

Chapter 5

Computational Experiments

The TDALT and TDCALT algorithms have been implemented and tested in practice, using two different road networks: the European road network, which is used as a common benchmark to compare with other algorithms in the literature, and the French road network, for which we have real time-dependent data available. We performed a large number of computational experiments in order to evaluate the impact of each component of our algorithm in determining the final speed. This gives us insight for the comparison with other existing algorithms. Then we designed experiments for the dynamic scenario, so to prove that our method can indeed be used for real-world applications.

The rest of this section is organized as follows. In Section 5.1 we provide details on the input data and the machines used for our experiments. In Section 5.2 we report experiments on the different contraction rates, and discuss the results. In Section 5.3 we provide a large experimental evaluation of the query algorithm with different parameters, compare it with existing algorithms, and analyse the different factors that contribute to its speed. Section 5.4 studies the dynamic scenario, and concludes this section.

5.1 Input data

We tested our algorithms on the road network of Western Europe provided by PTV AG for scientific use, which has 18 029 721 vertices and 42 199 587 arcs. A travelling time in uncongested traffic situation was assigned to each arc using that arc's category (13 different categories) to determine the travel speed. Time-dependent data for this instance is not available, thus we generated it (see Section 5.1.1). This road network is commonly used as a benchmark for routing algorithms, and it allows us to compare with the results reported in the literature. Tests on the US road network show very similar results for all the experiments, as confirmed by many works (e.g. (112; 46)), hence we do not report them.

The algorithms were also tested on another real-world instance, provided

by the Mediamobile company for our research, which models the road network of France alone. This graph comprises 7 265 051 vertices and 16 200 683 arcs; it is generated directly from the TeleAtlas 2006 Q1 data (116).

Our implementation is written in C++ using solely the STL. As priority queue we use a binary heap. For the European road network, our tests were executed on one core of an AMD Opteron 2218 running SUSE Linux 10.3. The machine is clocked at 2.6 GHz, has 16 GB of RAM and 2×1 MB of L2 cache. For the French road network, we used one core of an Intel Xeon X3553, clocked at 2.6 Ghz, with 16 GB of RAM and 2×6 MB of L2 cache. The program was compiled with GCC 4.1, using optimization level 3. Unless otherwise stated, we use 32 *avoid* landmarks (67), computed on the core of the input graph using the lower bounding function λ to weight edges, and we use the tightened potential function π_b^* (3.1) as potential function for the backward search, with 10 checkpoints.

5.1.1 Time-dependent arcs

Unfortunately, we are not aware of a *large* publicly available real-world road network with time-dependent arc costs. We therefore used artificially generated costs for the European instance. In order to model the time-dependent costs on each arc, we developed a heuristic algorithm, based on statistics gathered using real-world data on a limited-size road network; we used piecewise linear cost functions, with one breakpoint for each hour over a day. Arc costs are generated assigning, at each node, several random values that represent peak hour (i.e. hour with maximum traffic increase), duration and speed of traffic increase/decrease for a traffic jam; for each node, two traffic jams are generated, one in the morning and one in the afternoon. Then, for each arc in a node's arc star, a *speed profile* is generated, using the traffic jam's characteristics of the corresponding node, and assigning a random increase factor between 1.5 and 3 to represent that arc's slowdown during peak hours with respect to uncongested hours. We do not assign speed profile to arcs that have both endpoints at nodes with level 0 in a pre-constructed Highway Hierarchy (Section 1.4.4.1), because they are supposed have low importance in the road hierarchy. Excluding those arcs, we assign a speed profile to the remaining 5% most important edges in the graph, where importance is determined with the provided arc categories, and for each arc we use the traffic jam values associated with its endpoint with smallest ID. All arcs which are not assigned a speed profile have the same travelling time value throughout the day, equal to their static travelling time. This method to generate time-dependent speed profiles was developed to ensure spatial coherency between traffic increases, i.e. if a certain arc is congested at a given time, then it is likely that adjacent arcs will be congested too. This is a basic principle of traffic analysis (86).

For the French road network, the Mediamobile company provided real-world time-dependent data for 8374 arcs, in the form of piecewise linear functions

with one breakpoint for each hour of the day, and realistic travelling time over all non time-dependent arcs.

The breakpoints of speed profiles are stored in memory as a multiplication factor with respect to the speed in uncongested hours. The travelling time of an arc at time τ is computed via linear interpolation of the two breakpoints that precede and follow τ. The breakpoints are stored in an additional array, ordered by the edges they are assigned to. Similarly to an adjacency array graph data structure (37), each arc has a pointer to the first of its assigned breakpoints.

5.2 Contraction rates

In this section we analyse the effect of the contraction parameters on the performance of TDCALT and on the required preprocessing time and space. Results are reported in Table 5.1 for the European road network and in Table 5.2 for the French road network.

To measure the preprocessing effort, we report the percentage of nodes of the original graph which are not bypassed, i.e. nodes which are in the core, the time and additional space required by the preprocessing phase, the increase in number of edges and interpolation points of the merged graph with respect to the original graph. To analyse the speed of the shortest path computations with respect to the contraction parameters, we report the average size of the search space and CPU time to compute exact solutions and approximatd solutions to 10 000 random queries. For the approximated solutions, we fix the approximation constant of the algorithm K to 1.15, which experiments confirmed to be a good compromise between speed and quality of computed paths (see Section 5.3). As the performed queries may compute approximated results instead of optimal solutions when $K > 1$, we record three different statistics to characterize the solution quality: error rate, average relative error, maximum relative error. By *error rate* we denote the percentage of computed suboptimal paths over the total number of queries. By *relative error* on a particular query we denote the relative percentage increase of the approximated solution over the optimum, computed as $\omega/\omega^* - 1$, where ω is the cost of the approximated solution computed by our algorithm and ω^* is the cost of the optimum computed by Dijkstra's algorithm. We report *average* and *maximum* values of this quantity over the set of all queries. Note that contraction parameters of $C = 0.0$ and $H = 0$ yield a pure TDALT setup, i.e. there is no contraction, hence no hierarchy is build, and the TDALT algorithm is applied directly on the original graph (instead of applying it only on the core). We fix the maximum number of interpolation points for all shortcuts to $I = 200$, as suggested in (41), and analyse the performance of the algorithm as we vary the remaining contraction parameters: the maximum expansion of a bypassed node C, and the hop limit of added shortcuts H (see Chapter 4 for more details).

Core			Preprocessing				Exact Query		Approx. Query ($K = 1.15$)				
param.		core	time	space	increase in		#settled	time	error	relative error		#settled	time
C	H	nodes	[min]	[B/n]	#edges	#points	nodes	[ms]	rate	avg.	max	nodes	[ms]
0.0	0	100.0%	28	256	0.0%	0.0%	2 931 080	2 939.3	40.1%	0.303%	10.95%	250 248	188.2
0.5	10	35.6%	15	99	9.8%	21.1%	1 165 840	1 224.8	38.7%	0.302%	11.14%	99 622	78.2
1.0	20	6.9%	18	41	12.6%	69.6%	233 788	320.5	34.7%	0.288%	10.52%	19 719	21.7
2.0	30	3.2%	30	45	9.9%	114.1%	108 306	180.0	34.9%	0.287%	10.52%	9 974	13.2
2.5	40	2.5%	39	50	9.1%	138.0%	84 119	149.7	34.1%	0.275%	8.74%	8 093	11.4
3.0	50	2.0%	50	56	8.7%	161.2%	70 348	133.2	32.8%	0.267%	9.58%	7 090	10.3
3.5	60	1.8%	60	61	8.5%	181.1%	60 636	122.3	33.8%	0.280%	8.69%	6 227	9.2
4.0	70	1.5%	88	74	8.5%	223.1%	52 908	115.2	32.8%	0.265%	8.69%	5 896	8.8
5.0	100	1.2%	134	89	8.6%	273.5%	45 020	110.6	32.6%	0.266%	8.69%	5 812	8.4

Table 5.1: Performance of TDCALT for different contraction rates on the European road network. C denotes the maximum expansion of a bypassed node, H the hop-limit of added shortcuts. The third column records how many nodes have *not* been bypassed applying the corresponding contraction parameters. Preprocessing effort is given in time and *additional* space in bytes per node. We also report the increase in number of edges and interpolation points of the merged graph compared to the original input.

We focus on the European road network first (Table 5.1). As expected, increasing the contraction parameters has a positive effect on query performance. Interestingly, the space overhead first decreases from 256 bytes per node to 41 ($C = 1.0$, $H = 20$), and then increases again. The reason for this is that the core shrinks very quickly, hence we store landmark distances only for 6.9% of the nodes. On the other hand, the number of interpolation points for shortcuts increases by up to a factor ≈ 4 with respect to the original graph. Storing these additional points is expensive and explains the increase in space consumption. In particular, we observe that increasing the contraction parameters from $C = 0, H = 0$ to $C = 2.0, H = 30$ yields a reduction of the number of nodes in the core of a factor ≈ 33, while passing from $C = 2.0, H = 30$ to $C = 4.0, H = 70$ leads to a further reduction of only a factor ≈ 2; similarly, the search spaces for exact queries shrink by a factor ≈ 30 in the first case, but only by a factor ≈ 2 in the second case. There is the same behaviour for approximate queries. Thus, we clearly observe that there are diminishing returns when increasing the contraction parameters. Moreover, the number of interpolation points that have to be stored in memory rapidly becomes large and offsets the space saved by not storing landmark distances for non-core nodes. Preprocessing time increases as well, because performing operations on long shortcuts involves a large amount of floating point computations, and is as such very expensive. This justifies our choice of limiting the number of interpolation points of all shortcuts to $I = 200$, to avoid the explosion of required preprocessing space.

In terms of speed of the shortest path computations, increasing the contraction parameters always brings an advantage, albeit very small if C and H are large enough. For exact computations, TDCALT with contraction parameters $C = 5, H = 100$ yields a reduction of the average search space of up to a factor 65 with respect to TDALT (i.e., $C = 0, H = 0$). The reduction in terms of average CPU time is of a factor 26. This demonstrates the effectiveness of hierarchical speedup techniques on road networks, even in the time-dependent case.

The same behaviour can be observed for approximate queries, although the reduction in terms of the size of the search space is smaller. It is also interesting to note that if we allow more and longer shortcuts to be built, then the error rate decreases, as well as the maximum and average relative error. We believe that this is due to a combination of factors. First, long shortcuts decrease the number of settled nodes and have large costs, so at each iteration of TDCALT the key of the backward search priority queue β increases by a large amount. As the algorithm switches from phase 2 to phase 3 when $\mu/\beta < K$, and β increases by large steps, phase 3 starts with a smaller maximum approximation value for the current query μ/β. This is especially true for short distance queries, where the value of μ is small. Second, the core becomes very small for large contraction parameters. This increases the chance that the subpath of the shortest path which passes through the core has a small number of arcs (possibly, only

Core param. C	Core param. H	Core core nodes	Preprocessing time [min]	Preprocessing space [B/n]	Preprocessing increase in #edges	Preprocessing increase in #points	Exact Query #settled nodes	Exact Query time [ms]	Approx. Query ($K = 1.15$) error rate	Approx. Query relative error avg.	Approx. Query relative error max	Approx. Query #settled nodes	Approx. Query time [ms]
0.0	0	100.0%	13	256	0.0%	0.0%	136 202	81.1	15.1%	0.255%	15.88%	99 005	57.7
0.5	10	25.8%	5	70	12.3%	12.5%	45 449	29.5	12.2%	0.201%	12.96%	31 190	19.9
1.0	20	5.6%	10	41	12.6%	12.4%	10 351	8.2	9.4%	0.153%	12.60%	7 077	5.7
2.0	30	3.2%	196	11	9.1%	9.8%	5 491	4.9	7.7%	0.128%	13.63%	3 912	3.5
2.5	40	2.2%	272	9	8.0%	8.8%	4 449	4.1	6.9%	0.115%	11.62%	3 211	2.9
3.0	50	1.8%	307	7	7.3%	8.1%	3 815	3.5	7.4%	0.116%	12.75%	2 780	2.6
3.5	60	1.5%	307	6	6.8%	7.5%	3 753	3.4	6.8%	0.097%	12.05%	2 817	2.6
4.0	70	1.3%	307	5	6.4%	7.2%	3 484	3.1	6.4%	0.103%	13.13%	2 675	2.3
5.0	100	0.9%	310	4	5.6%	6.4%	3 988	3.2	6.6%	0.091%	13.43%	3 319	2.5

Table 5.2: Performance of TDCALT for different contraction rates on the French road network. Same column labels as in Table 5.1.

one); as shortcuts represent optimal distances, the chance of computing a sub-optimal path decreases. Summarizing, large contraction parameters require more preprocessing time and space, but yield better results in terms of size of the search space and query speed. On the other hand, experiments on the dynamic cost updates (Section 5.4) show that the length of shortcuts should be limited, if we want to perform cost updates in reasonable time.

Slightly different results are obtained on the French road network (Table 5.2). In this case, increasing contraction parameters always yields a decrease in required preprocessing space. The reason for this is that time-dependent arcs are found only in one region of the graph (namely, Paris and its surroundings), and the number of arcs is relatively small. Therefore, for increasing C and H we essentially contract more *static* arcs, which have only one associated breakpoint (the static arc travelling time). This does not bring the explosion of the number of breakpoints, as observed on the European road network. As a consequence, required preprocessing space keeps decreasing. We can also see that the percentage of nodes in the core is typically smaller with respect to the European road network with the same contraction parameters. We explain this by observing that the limit on the maximum number of interpolation points that would appear on new shortcuts does not apply on this road network, because the number of interpolation points is small. Hence, more contraction is carried out. The decrease in size of the search spaces and query times tails off after $C \geq 2.5$ and $H \geq 40$. The same applies for error rate and relative error.

5.3 Random Queries

In this section we analyse the performance of TDCALT for different values of the approximation constant K, using the European road network as input. In this experiment we used contraction parameters $C = 3.5$ and $H = 60$, i.e. we allow long shortcuts to be built so to favour query speed. We did not use larger values for the contraction parameters because the reduction in terms of CPU time is small, and the dynamic cost updates become unpractical (see Section 5.4). Results are recorded in Table 5.3 for the European road nework, and are gathered over 10 000 queries with source and destination nodes picked at random. For comparison, we also report the results on the same road network for the time-dependent versions of Dijkstra, unidirectional ALT, TDALT and the time-dependent SHARC algorithm (Section 1.4.6). In particular, Dijkstra's algorithm is used as a baseline to measure speedup factors, while SHARC currently represents the state-of-the-art for time-dependent shortest paths algorithms, although it is not able to deal with dynamic scenarios. Results for the French road network are reported in Table 5.5.

We report the amount of preprocessing time (in minutes) and space (in additional bytes per node) required by each algorithm. Besides, the performed

queries may compute approximated results instead of optimal solutions, depending on the value of K; thus, as in Section 5.2 we record three different statistics to characterize the solution quality: error rate, average relative error, maximum relative error. We recall that error rate is the percentage of computed suboptimal paths, and relative error on a particular query is the relative percentage increase of the approximated solution over the optimum. We report average and maximum values of the relative error over the set of all queries. We also record the average number of nodes settled at the end of the computation by each different algorithm, as well as the average CPU time in milliseconds.

In the following, we restrict ourselves to the scenario where only distances — not the complete paths — are required. However, our shortcut expansion routine for TDCALT (Section 4.2.3) needs less than 1 ms to output the whole path; the additional space overhead is ≈ 4 bytes per node.

We focus on the European road network first, which is the largest instance, hence the most interesting. In terms of preprocessing space, TDCALT with contraction parameters $C = 3.5, H = 60$ is the algorithm requiring less memory: only 61 additional bytes per node, while SHARC requires 118. Both TDALT and unidirectional ALT store landmark distances for all nodes in the graph, thus occupying 256 additional bytes per node. Algorithms which do not employ a hierarchical structure require a shorter preprocessing time: 28 minutes for TDALT and unidirectional ALT, which is the time to select 32 landmarks with the *avoid* heuristic and compute landmark distances. The contraction phase takes longer: 60 minutes for TDCALT, which only has to compute landmark distances for the core after the graph contraction, while SHARC takes 392 minutes because of the computation of arc-flags.

We now analyse the size of the search spaces and query times for the different algorithms. We restrict our attention to exact algorithms, i.e. $K = 1$. In our comparison, the algorithm with the largest average search space is Dijkstra's algorithm, with ≈ 8.8 millions nodes, and represents our baseline. The TDALT algorithm yields a reduction of a factor 3 with respect to the baseline. Interestingly, unidirectional ALT settles 4.31 fewer nodes with respect to Dijkstra's algorithm, thus it has a smaller search space than the bidirectional TDALT algorithm. This is easily explained if we consider that the bidirectional algorithm described in Chapter 3 may explore twice all nodes in the search space of the backward search. The SHARC algorithm only needs to settle 132.69 times fewer nodes than Dijkstra's algorithm, and TDCALT yields a further reduction, with a search space which is 145.76 smaller than the baseline. Thus, hierarchical methods are considerably more efficient in terms of number of settled nodes with respect to algorithms that only deal with a plain graph, as confirmed by many studies on static road networks (e.g. (19)).

If we consider the average number of settled nodes per millisecond, we obtain the following ranking:

1. Dijkstra's algorithm: 1688.6

Table 5.3: Performance on the European road network of time-dependent Dijkstra, unidirectional ALT, SHARC, TDALT and TDCALT with different approximation values K.

		PREPROC.		ERROR			QUERY	
		time	space		relative		# settled	time
technique	K	[min]	[B/n]	rate	av.	max	nodes	[ms]
Dijkstra	-	0	0	0.0%	0.000%	0.00%	8 877 158	5 757.4
uni ALT	-	28	256	0.0%	0.000%	0.00%	2 056 190	1 865.4
SHARC	-	392	118	0.0%	0.000%	0.00%	66 908	78.1
TDALT	1.00	28	256	0.0%	0.000%	0.00%	2 931 080	2 953.3
	1.05	28	256	3.4%	0.013%	4.16%	1 516 710	1 409.5
	1.07	28	256	7.0%	0.033%	6.82%	1 038 030	924.8
	1.10	28	256	19.6%	0.108%	7.88%	561 253	464.2
	1.12	28	256	29.3%	0.188%	10.52%	381 854	299.9
	1.15	28	256	40.1%	0.303%	10.95%	250 248	184.4
	1.20	28	256	48.5%	0.498%	12.05%	164 419	111.1
	1.25	28	256	51.0%	0.603%	21.64%	134 911	86.1
	1.30	28	256	52.0%	0.669%	21.64%	122 024	75.3
	1.35	28	256	52.6%	0.712%	21.64%	116 090	70.3
	1.50	28	256	52.8%	0.734%	21.64%	113 040	68.1
	1.75	28	256	52.9%	0.737%	30.49%	112 827	67.9
	2.00	28	256	52.9%	0.737%	30.49%	112 826	68.0
TDCALT	1.00	60	61	0.0%	0.000%	0.00%	60 961	121.4
	1.05	60	61	2.7%	0.010%	3.94%	32 405	62.5
	1.07	60	61	6.5%	0.030%	4.29%	22 633	42.1
	1.10	60	61	16.6%	0.093%	7.88%	12 777	21.9
	1.12	60	61	24.5%	0.158%	7.88%	9 132	14.9
	1.15	60	61	33.0%	0.259%	8.69%	6 365	9.2
	1.20	60	61	39.8%	0.435%	12.37%	4 707	6.4
	1.25	60	61	42.0%	0.549%	15.52%	4 160	5.4
	1.30	60	61	43.0%	0.611%	16.97%	3 943	5.0
	1.35	60	61	43.4%	0.649%	18.78%	3 843	4.9
	1.50	60	61	43.7%	0.679%	20.73%	3 786	4.8
	1.75	60	61	43.7%	0.682%	27.61%	3 781	4.8
	2.00	60	61	43.7%	0.682%	27.61%	3 781	4.8

2. unidirectional ALT: 1102.5

3. TDALT: 992.6

4. SHARC: 856.7

5. TDCALT: 502.1

As expected, Dijkstra's algorithm performs the smallest number of operations per node, hence it is able to settle more than 3 times the amount of nodes settled by TDCALT in the same amount of time. With respect to this criterion, the slowest algorithms are SHARC and TDCALT; both exploit an hierarchical setup, thus they have shortcuts. Therefore, both approaches need to relax more edges per node. In addition, since shortcuts have complicated cost function which are generated by operations on the original arc cost functions, their evaluation is computationally expensive, and explains the smaller number of settled nodes per millisecond. Moreover, bidirectional search introduces an additional overhead per node, as can be seen by comparing TDALT to unidirectional ALT. We suppose that this is due to the following facts: in the bidirectional approach, one has to check at each iteration if the current node has been settled in the opposite direction, and during phase 2 of the algorithm the upper bound μ has to be updated from time to time. The cost of these operations, added to the phase-switch checks, is probably not negligible.

If we only observe average query times for the different exact algorithms, we see that the fastest method is SHARC, which is 73.8 times faster than Dijkstra's algorithm. Second best is TDCALT, with a speedup of 47.6. Note that these speedup factors are significantly smaller than the search space reduction that the algorithms achieve, since Dijkstra's algorithm settles more nodes per unit of time. Unidirectional ALT is faster than TDALT: the speedups with respect to the baseline are, respectively, 3.1 and 1.95.

Next, we analyse the performance of TDALT and TDCALT when increasing the value of the approximation constant K. We immediately notice that the quality of the computed paths improves when using TDCALT with respect to TDALT for fixed K. As observed in Section 5.2, we believe that this is due to the presence of long shortcuts. The errors decrease in all respects: error rate, average relative error and maximum relative error. It is also interesting to note that the maximum relative error is very close to the theoretical allowed maximum (i.e. $K-1$) if K is small, whereas for larger values the approximation in practice is much smaller than the theoretical guarantee. When K is very large, phase 2 of the algorithm is very short and we immediately switch to phase 3, computing a path which passes necessarily through the first meeting point of the two search scopes. However, although this solution may be far from the optimum, it cannot be completely wrong, because the landmark potentials efficiently drive the two searchs towards their destination; thus, a path passing through the first meeting point is still a reasonable solution from a practical point of view. This explains why, even for large K, computed solutions are not suboptimal by a large factor. Search space sizes and query times greatly benefit from a value of K strictly larger than 1. In Figure 5.1 we compare query times of TDALT and TDCALT with respect to the value of the approximation constant. It is clear from the plot that a very small increase in the value of K initially brings a large saving in CPU time, whereas if K is already large enough then no further bene-

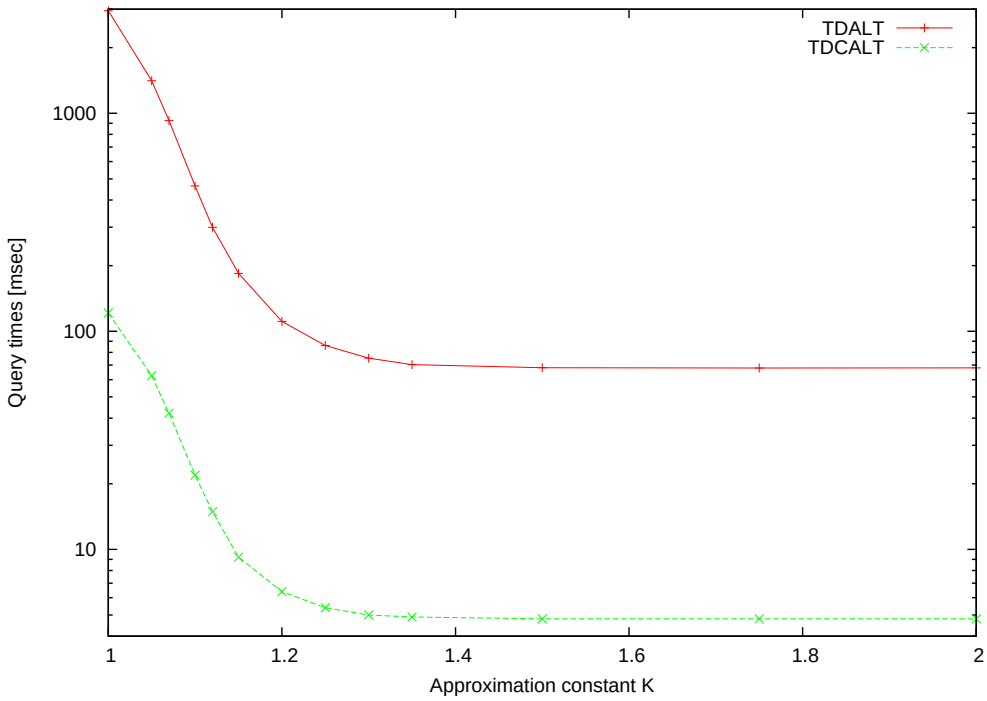


Figure 5.1: Comparison of the query times of TDALT and TDCALT with respect to the value of the approximation constant K. The y-axis has a logarithmic scale.

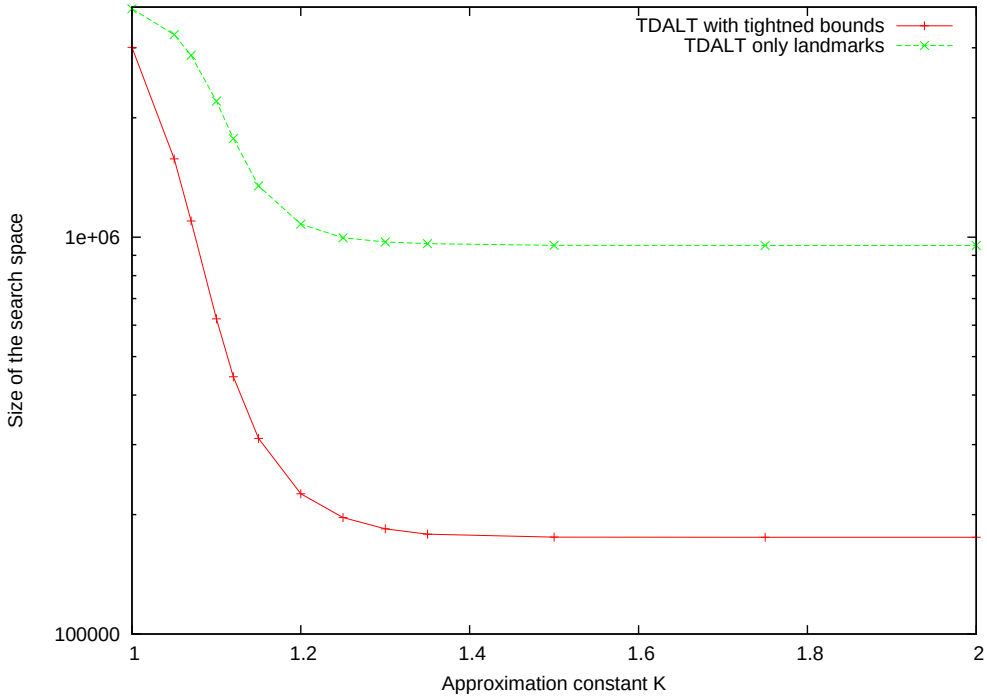


Figure 5.2: Comparison of the search space of TDALT with and without the tightened potential function π_b^* with respect to the value of the approximation constant K. The y-axis has a logarithmic scale.

Table 5.4: Performance on the European road network of time-dependent Dijkstra, unidirectional ALT, TDALT with and without the tightened potential function (3.1).

		PREPROC.		ERROR			QUERY	
		time	space		relative		# settled	time
technique	K	[min]	[B/n]	rate	av.	max	nodes	[ms]
TDALT	1.00	28	128	0.0%	0.000%	0.00%	3 009 320	2 842.0
	1.05	28	128	3.1%	0.012%	3.91%	1 574 750	1 379.2
	1.07	28	128	6.6%	0.034%	6.06%	1 098 470	915.4
	1.10	28	128	18.2%	0.106%	7.79%	622 466	481.9
	1.12	28	128	26.2%	0.181%	10.57%	444 991	325.0
	1.15	28	128	35.6%	0.292%	10.57%	311 209	214.2
	1.20	28	128	43.3%	0.485%	19.40%	225 557	145.3
	1.25	28	128	45.7%	0.589%	21.64%	196 581	122.3
	1.30	28	128	46.7%	0.655%	21.64%	184 143	111.6
	1.35	28	128	47.2%	0.703%	21.64%	178 410	107.4
	1.50	28	128	47.4%	0.722%	21.64%	175 468	105.3
	1.75	28	128	47.4%	0.725%	30.49%	175 248	105.3
	2.00	28	128	47.4%	0.725%	30.49%	175 247	105.4
TDALT$^-$	1.00	28	128	0.0%	0.000%	0.00%	3 763 990	3 291.6
	1.05	28	128	3.4%	0.023%	4.87%	3 238 120	2 683.5
	1.07	28	128	5.4%	0.046%	6.94%	2 874 500	2 290.7
	1.10	28	128	12.1%	0.122%	9.45%	2 201 870	1 619.2
	1.12	28	128	20.1%	0.237%	10.92%	1 772 080	1 218.4
	1.15	28	128	32.1%	0.474%	14.34%	1 345 930	842.0
	1.20	28	128	44.4%	0.787%	19.41%	1 079 290	618.3
	1.25	28	128	50.5%	0.993%	24.56%	996 631	553.2
	1.30	28	128	53.3%	1.104%	24.56%	972 294	531.5
	1.35	28	128	54.7%	1.166%	24.56%	962 950	524.7
	1.50	28	128	56.1%	1.248%	28.16%	953 704	526.4
	1.75	28	128	56.3%	1.261%	39.34%	952 947	518.2
	2.00	28	128	56.3%	1.262%	39.41%	952 933	518.9

fit is achieved. The best tradeoffs between path quality and speed are obtained for $K \in [1, 1.15]$.

Finally, we observe that TDCALT is at least one order of magnitude faster than TDALT on average. If we can accept a maximum approximation factor $K \geq 1.05$ then TDCALT is also faster than SHARC, by one order of magnitude for $K \geq 1.20$.

We now measure the impact of the tightened potential function π_b^* (3.1) on the search space reduction. To do so, we test on the European road network two different algorithms: TDALT with and *without* the tightened potential function.

Therefore, the latter will only use the landmark potential functions; we call this algorithm TDALT$^-$. In this setup, we used 16 landmarks generated with the *maxCover* heuristic (70). Results are reported in Table 5.4.

We observe that the tightened potential function reduces both the size of the search space and query times by a large factor. For exact queries, the average number of settled nodes is reduced by a factor 1.25, and query times by a factor 1.16. As we increase K, the impact of the π_b^* rapidly becomes larger: for $K = 1.1$, the search space reduction is of a factor 3.53, 4.39 for $K = 1.15$, up to a maximum of 5.43 for $K = 1.5$ and above. Figure 5.2 plots the size of the search space of TDALT and TDALT$^-$ with respect to the value of K, and supports our analysis. A similar behaviour is observed for query times. Moreover, the error rate decreases when using the tightened potential function, as well as both average and maximum error rate. This is not a trivial result: since TDCALT$^-$ explores a significantly larger amount of nodes, one would expect that it would find better solutions. This observation leads us to think that the tightened potential function not only reduces the number of settled nodes in general, but also prunes the search at nodes which cannot appear on good solutions, therefore yielding a more efficient exploration of the graph.

We now consider results obtained on the French road network, and reported in Table 5.5. We note that all reported values are subject to very small changes, if any at all, for $K \geq 1.10$. We give the following explanation: since the majority of arcs in this instance is static, the cost lower bounds given by λ which are used to compute landmark distances are equal to the costs used through the shortest paths computations, leading to near-optimal performance of landmark potentials. Therefore, there is not much room for improvement in terms of size of the search space. The only value which is affected by large values of K is the maximum relative error: the search stops sooner by a small amount of nodes, as indicated by the average size of the search space, but the maximum error increases by significant amounts. This suggests that important nodes are sometimes skipped during the search. Overall, it does not seem profitable in practice to set $K \geq 1.10$.

In terms of query speed, Unidirectional ALT is one order of magnitude faster than Dijkstra's algorithm. TDALT yields an additional CPU time saving of a factor two. TDCALT is on average one order of magnitude faster than TDALT, therefore almost three orders of magnitude faster than Dijkstra's algorithm. Similar considerations can be done with respect to the size of the search spaces. Error related values also decrease when switching from TDALT to TDCALT, as observed on the European instance. Summarizing, the analysis carried out on the road network of Western Europe is also valid for the road network of France, with the exception of the different behaviour of preprocessing time and space – which is discussed in Section 5.2.

Table 5.5: Performance on the French road network of time-dependent Dijkstra, unidirectional ALT, TDALT and TDCALT with different approximation values K.

		PREPROC.		ERROR			QUERY	
		time	space		relative		# settled	time
technique	K	[min]	[B/n]	rate	av.	max	nodes	[ms]
Dijkstra	-	0	0	0.0%	0.000%	0.00%	3 615 670	1 723.4
uni ALT	-	13	256	0.0%	0.000%	0.00%	290 178	140.5
TDALT	1.00	13	256	0.0%	0.000%	0.00%	136 202	81.4
	1.05	13	256	14.1%	0.166%	4.96%	100 884	59.0
	1.07	13	256	14.6%	0.205%	7.00%	100 013	58.4
	1.10	13	256	15.0%	0.234%	9.89%	99 303	58.0
	1.12	13	256	15.0%	0.243%	12.00%	99 156	57.9
	1.15	13	256	15.1%	0.255%	14.02%	99 062	57.8
	1.50	13	256	15.2%	0.260%	32.44%	98 970	57.7
	2.00	13	256	15.2%	0.260%	32.44%	98 969	57.7
TDCALT	1.00	307	6	0.0%	0.000%	0.00%	3 790	3.5
	1.05	307	6	6.0%	0.064%	4.89%	2 876	2.6
	1.07	307	6	6.5%	0.085%	6.36%	2 848	2.6
	1.10	307	6	6.8%	0.107%	9.55%	2 832	2.5
	1.12	307	6	6.9%	0.114%	10.07%	2 830	2.5
	1.15	307	6	7.0%	0.119%	12.43%	2 828	2.5
	1.50	307	6	7.1%	0.122%	16.62%	2 826	2.5
	2.00	307	6	7.1%	0.122%	16.62%	2 826	2.5

5.3.1 Local Queries

For random queries, TDCALT is one order of magnitude faster than TDALT on average. TDCALT is significantly faster than unidirectional ALT, while TDALT is faster than the unidirectional version of the algorithm only for $K \geq 1.05$. In order to gain insight whether these speedups derive from small or large distance queries, Fig. 5.3 reports the query times with respect to the Dijkstra rank, obtained on the European road network. For an s-t query, the Dijkstra rank of node t is the number of nodes settled before t is settled: thus, it is some kind of distance measure. These values were gathered on the European road network instance, using contraction parameters as in Table 5.3, i.e. $c = 3.5$ and $h = 60$.

Note that we use a logarithmic scale due to the fluctuating query times. The figure clearly indicates that both speedup techniques pay off only for long distance queries. If the source and destination node are close to each other, then unidirectional ALT is faster than TDCALT by an order of magnitude in some cases. This is expected, since for small distances TDCALT may result in a simple application of Dijkstra's algorithm, with no speedup techniques. For sufficiently long distances, however, the median of TDCALT is almost two orders

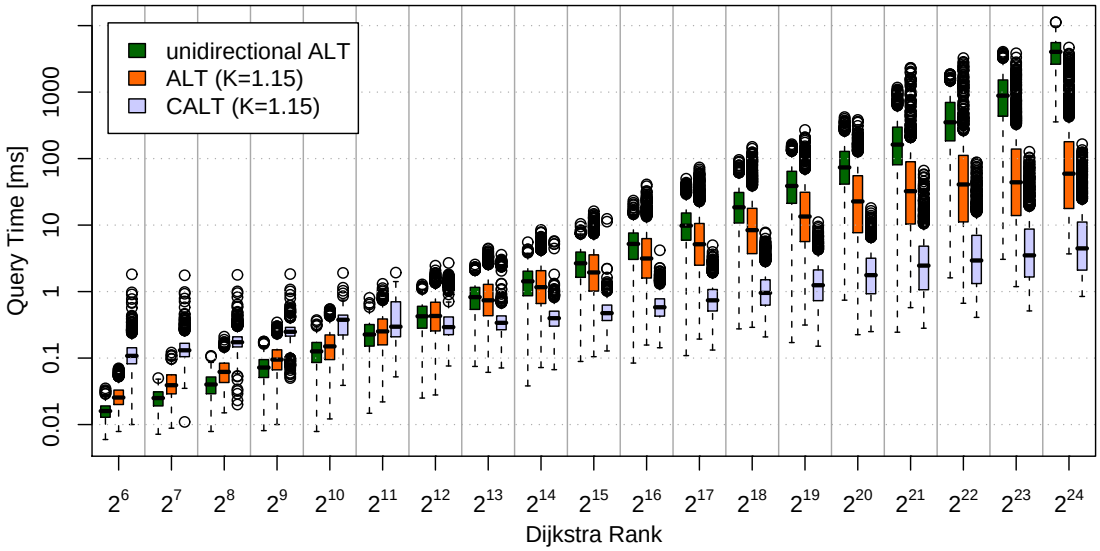


Figure 5.3: Comparison of unidirectional ALT, TDALT and TDCALT using the Dijkstra rank methodology (124). The results are represented as box-and-whisker plot: each box spreads from the lower to the upper quartile and contains the median, the whiskers extend to the minimum and maximum value omitting outliers, which are plotted individually.

of magnitude faster than unidirectional ALT. TDALT is typically positioned between unidirectional ALT and TDCALT: for small distance queries it is faster than TDCALT because it does not have to resort to a plain Dijkstra search, but for large distances it is one order of magnitude slower. It also interesting to note that some outliers of TDALT may be slower than unidirectional ALT for large Dijkstra ranks, whereas outliers of TDCALT are always faster than the median of unidirectional ALT.

Summarizing, the proposed speedup techniques are particularly effective for long distance queries, which are the most difficult cases to deal with in practice, hence the most interesting.

5.4 Dynamic Updates

In order to evaluate the performance of the core update procedure (see Section 4.3) we generated several traffic jams as follows: for each traffic jam, we select a path in the network covering 4 minutes of uncongested travel time on motorways. Then we randomly select a breakpoint between 6AM and 9 PM, and for all edges on the path we multiply the corresponding breakpoint value by a factor 5. As also observed in (46), updates on motorway edges are the most difficult to deal with, since those edges are the most frequently used during the shortest path computations, thus they contribute to a large number of shortcuts.

cont. C	cont. H	limit [min]	space [B/n]	traffic jam single[ms] av.	traffic jam single[ms] max	traffic jam batch[ms] av.	traffic jam batch[ms] max	single breakpoint single[ms] av.	single breakpoint single[ms] max	single breakpoint batch[ms] av.	single breakpoint batch[ms] max	query [ms]
0.0	0	–	256	0.0	0	0	0	0.0	0	0	0	188.2
0.5	10	5	123	0.4	28	372	488	0.1	5	97	166	81.5
		10	121	0.7	49	619	799	0.1	12	183	383	85.2
		15	119	0.7	49	707	1 083	0.1	11	202	407	74.2
		20	119	0.7	49	820	1 200	0.2	59	291	459	73.8
1.0	20	5	82	7.8	229	7 144	8 090	1.8	78	1 853	2 041	34.5
		10	72	21.2	778	20 329	22 734	5.8	371	5 957	9 266	27.1
		15	68	32.1	2 226	27 327	33 313	7.2	427	7 291	11 522	25.4
		20	66	37.0	2 231	30 787	39 470	8.8	1 197	8 476	11 426	22.8
2.0	30	5	88	17.4	290	16 293	17 493	5.7	283	5 019	6 017	33.7
		10	82	90.5	3 868	79 092	85 259	27.6	1 894	24 943	27 501	22.8
		15	79	171.0	4 604	120 018	142 455	49.4	2 451	46 237	58 936	19.7
		20	77	219.7	5 073	187 595	206 569	63.3	5 510	60 940	65 954	16.4

Table 5.6: CPU time required to update the core for different contraction parameters and limits for the length of shortcuts.

In Table 5.6 we report average and maximum required time over 1000 runs to update the core in case of a single traffic jam, applying different contraction parameters. We also report the corresponding figures for a batch update of 1000 traffic jams (computed over 100 runs), in order to reduce the fluctuations and give a clearer indication of required CPU time when performing multiple updates. Besides, we measured the average and maximum time required to update the core when modifying a single breakpoint on a motorway arc selected uniformly at random; we also record the corresponding values when modifying 1000 single breakpoints on random motorway arcs (computed over 100 runs). As there is no spatial locality when updating a single breakpoint over random arcs, this represents a worst-case scenario. Note that in this experiment we limit the length of shortcuts in terms of uncongested travel time (as reported in the third column). This is because in the dynamic scenario the length of shortcuts plays the most important role when determining the required CPU effort for an update operation, and if we allow the shortcuts length to grow indefinitely we may have unpractical update times. Hence, we also report preprocessing space in terms of additional bytes per node, and query times with $K = 1.15$. We remark that Table 5.6 only considers the CPU time required to update the core, and does not take into account the computational effort to modify the cost functions for arcs at level 0 in the hierarchy, i.e. not belonging to the core. However, this effort is negligible in practice, because the modification of a breakpoint of an arc outside the core has an influence only on the arc itself. Therefore, the update is carried out by simply modifying the corresponding breakpoint value, whereas the core update is considerably more time-consuming (see Section 4.3).

As expected, the effort to update the core becomes more expensive with increasing contraction parameters. First, we consider the scenario where we generate 1000 traffic jams over motorway arcs, and modify the cost functions accordingly. For $C = 0.5, H = 10$ the updates are very fast, even if we allow long shortcuts (i.e. 20 minutes of uncongested travel time). The average CPU time for an update of 1000 traffic jams is always smaller than 1 second, therefore we are able to deal with a large number of breakpoint modifications in a short time. This is confirmed by the very small average time required to update the core after modifying a random breakpoint on a random motorway arc, which is smaller than 0.2 milliseconds. As we increase the contraction parameters, dynamic updates take longer to deal with. A larger number of long shortcuts is created, therefore update times grow rapidly, requiring several seconds. The average time to update the core after adding 1000 traffic jams with contraction parameters $C = 1, H = 20$ is at least one order of magnitude larger than the respective values with parameters $C = 0.5, H = 10$. Very large updates are feasible in practice only if we limit the length of shortcuts to 5 minutes of uncongested travel time; for most practical applications, however, updates are not very frequent, therefore adding 1000 traffic jams in ≈ 30 seconds is reasonably fast. If

we consider contraction parameters $C = 2, H = 30$, then the updates for this scenario may require several minutes; however, limiting the length of shortcuts helps.

Next, we analyse update times for modifications of a single breakpoint over random motorway arcs. We observe that they confirm the analysis for the previous scenario (adding 1000 traffic jams). For small contraction parameters (or if we limit shortcuts to a small length in terms of uncongested travelling time), updating the core after modifying one breakpoint requires on average less than 10 milliseconds, whereas if we modify 1000 breakpoints we need less than 10 seconds. For $C = 0.5, H = 10$ we can carry out the updates in less than 0.5 seconds. If we allow shortcuts to grow, then updates may require several seconds.

If we compare the time required to update the core after adding 1000 traffic jams with respect to modifying 1000 breakpoints, we see that our update routine greatly benefits from spatial locality of the modified arcs: the first scenario is only ≈3-4 times slower than the second, but the number of modified arcs is larger, because each traffic jam extends over several motorway arcs. However, this is expected: as each shortcut is updated only once, modifications on contiguous arcs may require no additional effort, if all modified arcs belong to the same shortcut. In real world applications, traffic jams typically occur on contiguous arcs (86), therefore our update routine should behave better in practice than in worst case scenarios.

Summarizing, we observe a clear trade off between query times and update times depending on the contraction parameters, so that for those applications which require frequent updates we can minimize update costs while keeping query times < 100 ms, and for applications which require very few or no updates we can minimize query times. If most of the arcs have their cost changed we can rerun the core arcs computation, i.e. recomputing all arcs on the core from scratch, which only takes a few minutes.

Chapter 6

A Real-World Application

One of the main objective of this thesis was to develop an algorithm capable of answering several shortest path queries per second, so that a real-time path computing service taking into account both real-time traffic and traffic forecastings could be proposed on the website of the Mediamobile company. In this section we describe the existing industrial platform of this company, and the architecture of a new platform which integrates the algorithm described above with the existing components.

The rest of this section is organized as follows. In Section 6.1 we briefly describe the existing components in the architecture where we wish to integrate our implementation of the TDCALT algorithm. In Section 6.2 we give a description of how this integration was carried out. In Section 6.3 we give a mathematical formulation for the problem of computing the updated cost function breakpoint values, so as to adapt the function to real-time traffic information and forecastings.

6.1 Description of the existing architecture

Mediamobile is a company whose primary purpose is to gather, aggregate and redistribute real-time traffic information. Several services are built on top of this basic function. The main component of their architecture is the traffic information server SDT (*Serveur De Trafic*, "traffic server" in French), which gathers real-time traffic information from various sources and treats them so as to make them directly utilizable for other services. In particular, the SDT is also the provider of traffic forecast information, and, as such, is the only source of traffic information which is necessary for our needs.

Real-time traffic information is obtained through different means: the most reliable information is measured directly on-site by local administrations, usually by means of electromagnetic loops or cams that are capable of determining the speed of passing vehicles. Another important source is represented by Float-

ing Car Data: cars driving in a fleet are equipped with GPS devices that constantly monitor their position and speed, therefore providing information on the congestion status of road segments that some vehicles have recently gone through. Real-time traffic information is updated every few minutes (tipically, two minutes). Real-time traffic information is then used to feed different prediction models, so that traffic forecasts are generated. Forecasts are divided into three different categories, depending on the time horizon after the prediction is made that they address:

1. short-term forecasts: a few seconds to 10-20 minutes,
2. mid-term forecasts: 10-20 minutes to 4-5 hours,
3. long-term forecasts: 4-5 hours to several months.

Long-term forecasts rest on aggregated historical data, and represent the speed profiles which were referred to throughout this thesis. Short- and mid-term forecasts are updated every few minutes, as a consequence of changes in the real-time traffic information, whereas long-term forecasts may be updated once every several months or even years, thus they can be considered as static information.

The SDT provides an API that allows other services to query and obtain traffic information; among these services, the one that is relevant for this thesis is the one that provides the main features of the site `www.v-trafic.com`, which allows web users to see the real-time traffic status over all France. This service runs on a Windows machine and exposes a web service towards the web site. In the following, we will refer to the C# application run by this machine as the *paths computation service*. A screenshot of the web site with the route planning module can be found in Figure 6.1. Note that, at the moment of writing this thesis, the only front-end for the integrated path computation platform described in this section is the web site, but it would be easy to allow any device with internet connection capabilities to query the platform. In particular, this could be implemented for GPS devices, so as to allow an internet-connected service where the GPS need not know real-time traffic information and forecasts over the whole network: it suffices to send a path query to the path computation platform, which utilizes all the available traffic information up-to-date, and display the path which answers the query. Our objective is to integrate our C++ implementation of TDCALT with the C# paths computation service, so as to provide users with the possibility of calculating the optimal path between two points. The paths computation service has an internal graph representation of the French road network, since it must be able to project points on the road network (i.e. find the node which is closer to given cohordinates) and draw on the map, in order to indicate the traffic congestion status on monitored road segments and highlight the computed path after a shortest path query. These capabilities are implemented using the Google Maps API (73). Moreover, this

Figure 6.1: The Mediamobile web site with the route planning module. The screenshot has been taken on the beta version of the web site, which employs the TDCALT algorithm through the platform discussed in this section.

machine is directly connected to the SDT and refreshes real-time traffic information and traffic forecasts after every update of the SDT. The paths computation service also implements a time-dependent Dijkstra's algorithm which uses the most accurate real-time and forecast traffic information available; however, this implementation may take several seconds of CPU time if applied on the whole French road network, thus it is not suitable for answering user queries in real-time. In the following, we will refer to the C# application run by this machine as the *paths computation service*.

6.2 Description of the proposed architecture

We describe now the architecture that we implemented to integrate path computing capabilities using TDCALT within the existing platform.

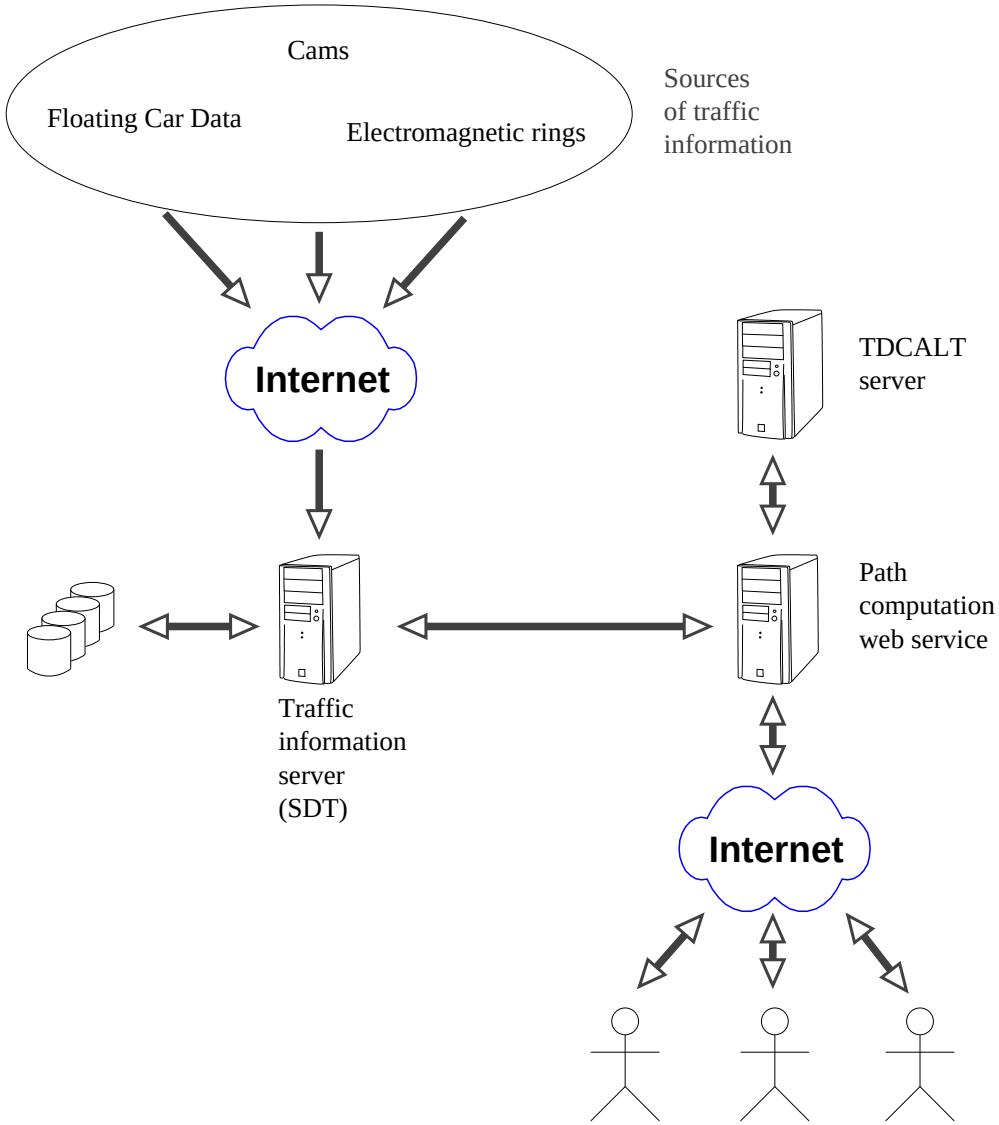


Figure 6.2: Schematic representation of the proposed architecture.

One of the first difficulties that we had to deal with is the fact that the existing paths computation service is coded in C# and runs on Windows, whereas our TDCALT implementation is coded in C++ and runs on Linux. Thus, both applications could not run on the same machine. We decided to run the TDCALT implementation on a separate machine, which is only accessible through the paths computation service, and whose only purpose is to compute the paths between two points with starting time at a given time of the day. A sketch of this architecture can be seen in Figure 6.2.

The TDCALT server has an internal graph representation which uses different node and arc identifiers; the reason behind this is that identifiers provided

by mapping companies (e.g. NavTeq, TeleAtlas) are typically unpractical to deal with, due to excessive length, whereas our C++ implementation needs integer identifiers for both arc and nodes which are contiguous and start from zero. Thus, we compute a mapping between the graph representation available on the paths computation service and the one used by the TDCALT server. The paths computation service in C# is responsible for the following tasks:

- expose a web service towards the web site;
- obtain the updated traffic information from the SDT, process it and send it to the TDCALT server;
- translate external arc/node identifiers into the identifiers used by the TDCALT server, and vice-versa;
- manage the queue of pending queries;
- compute and display the road map on the web site after a path has been computed.

The processing of updated traffic information, i.e. the update of the coefficients of the piecewise linear cost functions, is described in more detail in Section 6.3. Communication towards and from the TDCALT server is carried out through TCP/IP with binary encoded messages.

The TDCALT server accepts incoming TCP packets on two different ports. The first one is dedicated to the computation of shortest paths: a query packet contains the internal identifier of the source and destination nodes, as translated by the C# paths computation service through the available mapping, and the departure time; the machine carries out the calculations, and returns back the list of arcs on the shortest path, as well as its cost. Shortest paths computations cannot be done in parallel, since the nodes are labeled with one of the different statuses: `unreached, explored, settled` (see Section 1.4.2), and for each query the labels must be initialized. The second port is dedicated to the cost function updates. Whenever the cost function should be updated, the C# service sends to the TDCALT server the data containing arc identifiers and new breakpoint values through this port. When an update is pending, the TDCALT server does not accept path queries anymore; if there is a shortest path computation in progress, then that computation is carried out. As soon as there are no path computations in progress, the update phase starts: the set of all shortcuts that must be updated is split equally among all available CPUs, and the procedure discussed in Section 4.3 is applied. Since each shortcut can be updated independently from the others, this routine can be run in parallel with no additional effort. When the update procedure is complete, the path computations resume normally.

Additionally, we run an instance of the TDCALT server *without* real-time traffic information. This serves the purpose of answering shortest paths queries whose departure time is not in the near future with respect to the instant at which the query is received. Typically, this happens when users want to compute a shortest route using statistical information for a particular day of the week. To this end, Mediamobile provides 4 different speed profiles, that correspond to different combinations between subsets of days in the week and holiday/working day. We run an instance of the TDCALT which manages all 4 profiles, and answers to shortest paths queries using the most appropriate one.

The TDCALT server is run on a machine equipped with a Quad-Core AMD Opteron 2354 processor and 16 GB of RAM. Summarizing, this machine runs an instance of the server with real-time traffic information on one core, and an instance without real-time traffic information and 4 different speed profiles on another core; as these instances are run separately, they can be used in parallel. The first instance takes approximately 1.6 GB of RAM, the second one 4 GB. The two remaining CPU cores are used only for the profile update.

6.2.1 Load balancing and fault tolerance

We implemented a simple fault tolerance scheme to increase the availability of the path computation service. At the same time, this scheme provides a load balancing mechanism. The scheme is illustrated in Figure 6.3, and works as follows. Both the machine that runs the C# path computation service and the C++ TDCALT server are duplicated, so that there are two perfectly equal platforms to answer a shortest path query. A load balancer is put in the layer between users (e.g. the website) and the C# path computation service, and dispatches requests to the machine with the smallest work load. Similarly, a load balancer represents the layer between the two machines that run che C# path computation service and the C++ TDCALT server. Requests are dispatched to the first available machine. The two load balancers also implement a fault tolerance scheme: whenever one of the queried machines is down, queries are automatically sent to its duplicate. All messages are exchanged as short TCP packets and each TCP connection is valid for one shortest path query only; therefore, the load balancer can connect a C# path computation service to a C++ TDCALT server by simply forwarding the TCP packet containing the query. Note that, in terms of performance, it would be better to keep the connection between the C# path computation server and the TDCALT server always open, so that we do not have to waste resources on establishing the connection. However, with such a scheme we would not be able to implement the fault tolerance and load balancing mechanism.

It can be seen that this architecture provides a natural way to deal with real-time traffic information updates: whenever one of the two real-time TDCALT servers is running an update (and is therefore unvailable for shortest paths com-

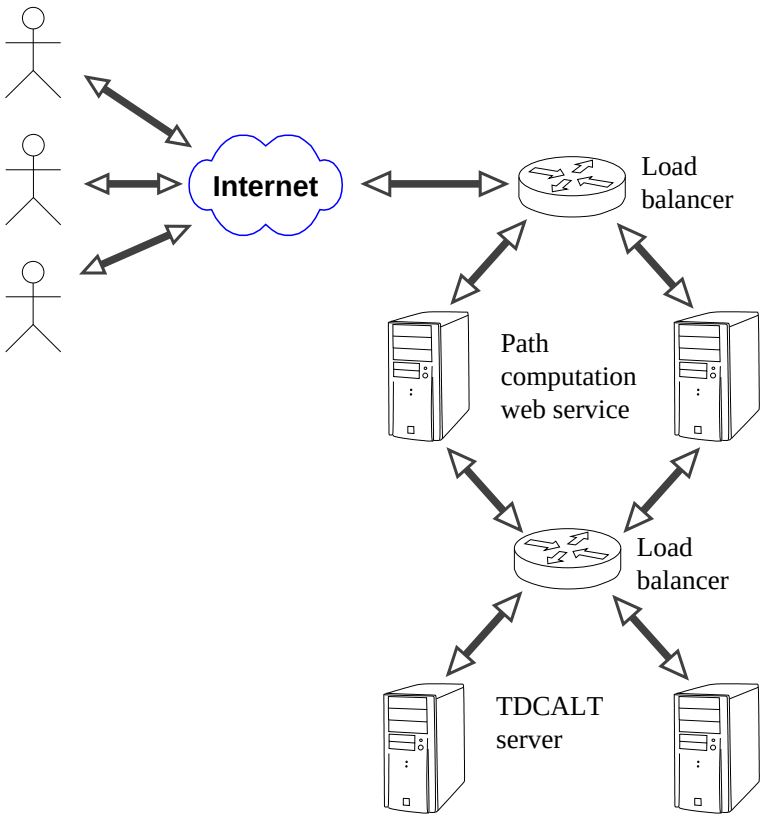


Figure 6.3: Schematic representation of the load balancing and fault tolerance scheme.

putations), the load balancer forwards all queries to the duplicate machine, which will update the profiles only when the first one has finished the update. This way, at least one of the two machines is always available.

The load balancer is able to tell the status of each TDCALT server by querying a monitoring service via TCP. The monitoring service is a thread spawned by the main TDCALT process, which therefore shares the same address space and is able to detect if the main program has encountered errors. The monitoring thread listens for TCP connection on a different port than the main TDCALT process, and, whenever it receives a message, sends back one of the following two-characters ASCII encoded strings:

1. `OK`: the server is alive and working;

2. `KO`: the server is alive but has encountered problem that prevent it from answering to queries (e.g. network related problems);

3. `UP`: the server is alive but is currently updating profiles.

Obviously, if the monitoring thread does not answer or does not accept the connection, the main TDCALT process is considered to be dead or unreachable. Since the load balancer queries the TDCALT server status every second, it may take up to one second to propagate the status information to the load balancer. Therefore, when a TDCALT server wants to perform an update of its cost functions, it first sets its status to `UP`, then waits for one second (while still accepting and answering incoming path queries), and finally starts the update. This way we avoid sending shortest path queries to a TDCALT server which is performing an update.

6.3 Updating the cost function coefficients

Through this section, we assume that for each time instant there is no overlapping traffic information, i.e. we only have one of the following:

1. real-time information,
2. short-term forecasts,
3. mid-term forecasts,
4. long-term forecasts,

taken in order of accuracy. For a given arc, each information is represented as point $(\tau, f(\tau))$ where τ represents the time instant and $f(\tau)$ is the travelling time over that arc starting the traversal at time τ. Whenever one of the different sources of traffic information is updated, we would like to compute the piecewise linear function that best fits the new information, so that the corresponding arc cost function can be updated. Since this task must be carried out often, it should have a small computational cost. Hence, it is natural to solve this problem using a least squares estimation of the cost function. We state the problem formally as follows.

We are given a set of points $(x_i, y_i), i = 1, \ldots, n$ where $x_i \in [0, P]\ \forall i = 1, \ldots, n$ and a set of breakpoint positions $r_1 = 0, \ldots, r_{m+1} = P$ with $r_i \in [0, P]\ \forall i = 1, \ldots, m+1$ such that $[0, P]$ can be partitioned into m intervals $[r_i, r_{i+1}]\ i = 1, \ldots, m$. We want to compute a continuous piecewise linear function $f : [0, P] \to \mathbb{R}$,

$$f(x) = a_i x + b_i \text{ if } x \in [r_i, r_{i+1}), \tag{6.1}$$

periodic of period P, that minimizes the sum of the quadratic errors:

$$\Delta = \sum_{i=1}^{n} (f(x_i) - y_i)^2. \tag{6.2}$$

In order for f to be continuous, the condition

$$a_i r_{i+1} + b_i = a_{i+1} r_{i+1} + b_{i+1} \tag{6.3}$$

must hold $\forall i = 1, \ldots, m-1$; since f is periodic, we also have

$$b_1 = a_m r_{m+1} + b_m. \tag{6.4}$$

Constraints (6.3) allow us to express b_i for $i = 2, \ldots, m$ in terms of the remaining variables, while (6.4) expresses a_m in terms of b_1 and b_m. With simple algebraic calculations we obtain

$$b_i = \sum_{j=2}^{i} (a_j + a_{j-1}) r_j + b_1 \quad \forall i = 2, \ldots, m, \tag{6.5}$$

and

$$a_m = -\frac{\sum_{j=2}^{m} (a_j + a_{j-1}) r_j}{P}. \tag{6.6}$$

Thus, we need to optimize over the m variables $b_1, a_1, \ldots, a_{m-1}$. This is a convex problem which can be solved by vanishing the derivatives of (6.2) with respect to the variables. We obtain:

$$\begin{aligned}
\frac{d\Delta}{db_1} &= \sum_{x_j \in [0, r_2]} (a_1 x_j + b_1 - y_j) + \sum_{i=2}^{m} \sum_{x_j \in [r_i, r_{i+1}]} (a_i x_j + \sum_{h=2}^{i} (a_h + a_{h-1}) r_h + b_1 - y_j) \\
\frac{d\Delta}{da_1} &= \sum_{x_j \in [r_1, r_2]} (a_1 x_j + b_1 - y_j) x_j + \\
&\quad + \sum_{i=2}^{m} \sum_{x_j \in [r_i, r_{i+1}]} (a_i x_j + \sum_{h=2}^{i} (a_h + a_{h-1}) r_h + b_1 - y_j)(r_2 + r_1) \\
&\vdots \\
\frac{d\Delta}{da_k} &= \sum_{x_j \in [r_k, r_{k+1}]} (a_k x_j + \sum_{h=2}^{k} (a_h + a_{h-1}) r_h + b_1 - y_j) x_j + \\
&\quad + \sum_{i=k+1}^{m} \sum_{x_j \in [r_i, r_{i+1}]} (a_i x_j + \sum_{h=2}^{i} (a_h + a_{h-1}) r_h + b_1 - y_j)(r_{k+1} + r_k) \\
&\vdots \\
\frac{d\Delta}{da_{m-1}} &= \sum_{x_j \in [r_{m-1}, r_m]} (a_{m-1} x_j + \sum_{h=2}^{m-1} (a_h + a_{h-1}) r_h + b_1 - y_j) x_j + \\
&\quad + \sum_{x_j \in [r_m, P]} (-\frac{\sum_{h=2}^{m} (a_h + a_{h-1}) r_h}{P} x_j + \sum_{h=2}^{m} (a_h + a_{h-1}) r_h + b_1 - y_j)(P + r_{m-1}).
\end{aligned}$$

Solving this $m \times m$ linear system yields the optimal coefficients $a_i, b_i \; i = 1, \ldots, m$, which define the updated cost function (6.1).

Not all coefficients $a_i, b_i \; i = 1, \ldots, m$ must be modified at each update step. The main purpose of this procedure is to take into account real-time traffic information and forecasts in order to provide a reliable path to those users which are interested in computing a path with departure time close to the time instant at which the path is queried (via the web service). Obviously, it does not make sense to request paths with departure time in the past. Thus, we only update coefficients "in the future" with respect to the time at which the update is made. Formally, we are given a set of points $(x_i, y_i), i = 1, \ldots, n$ such that $(f(x_i) - y_i)/f(x_i) \geq T$, where T is a given threshold; this means that the current piecewise linear function $f(x)$ does not fit the points $(x_i, y_i), i = 1, \ldots, n$, hence we need to update its coefficients. Let $[r_L, r_U]$ be an interval with endpoints defined as: $L = \arg\max_{j \in \{1,\ldots,m+1\}} \{r_j | r_j \leq x_i, i = 1, \ldots, n\}, U = \arg\min_{j \in \{1,\ldots,m+1\}} \{r_j | r_j \geq x_i, i = 1, \ldots, n\}$. Then we update the coefficients a_i, b_i only in $[r_L, r_U]$, that is, we compute and modify only $a_i, b_i \; i = L, \ldots, U$. It is straightforward to obtain the modified values of the breakpoints from the variables $a_i, b_i \; i = L, \ldots, U$, so that they can be directly used as input for the update procedure (Section 4.3).

The update algorithm described in Section 4.3 requires the new breakpoint values to be larger than the ones used during the preprocessing phase. However, if we use the long-term traffic forecast information (i.e. speed profiles) to derive cost functions, then we are not guaranteed that the breakpoints can only increase with respect to their initial values. To obviate this problem, during the preprocessing phase we use specially computed time-dependent cost functions which represent, over an arc, the minimum travel time observed on that arc over one year of aggregated historical data. These functions provide a reasonable guarantee that the breakpoint values always stay above their initial values. In case the piecewise linear function fitting procedure computes a breakpoint value which is smaller than the initial value, then we use the latter. For those arcs where historical data is not available, we use a constant time-dependent cost function equal to the lower bounding function λ for all time instants. The drawback of this approach is that the lists of shortcuts which may be affected by the change on an arc (and thus need be recomputed) may grow (see Section 4.3), but since the update procedure is run in parallel on several CPUs the gain by parallelism can offset the loss due to the increased number of recomputed shortcuts, and keep the CPU time required for an update down to reasonable values (two seconds at most).

Part II

Mathematical Formulation Based Methods

The algorithms presented in Part I are able to find solutions to the TDSPP in a few milliseconds, even on continental sized road networks; however, the price for this rapidity are the restrictive assumptions on the form of the time-dependent arc cost functions, which must be piecewise linear and satisfy the FIFO property (Section 1.2.1). Moreover, we do not take into account additional constraints, such as prohibited turnings. From an industrial point of view, these problems are largely offset by the speed of the calculations, which makes real-time applications possible. However, we are also interested in the study of networks with no restrictions on the cost functions, possibly with some additional constraints that should be satisfied by the solution. This study has several purposes; mainly, it is useful to analyse the practical relevance of solutions obtained by lifting all the restrictions applied in Part I. We presented a mathematical programming formulation of the TDSPP in Section 1.3.2. In this chapter, we will study general-purpose algorithms towards the solution of Mixed-Integer Linear Programs and Mixed-Integer Nonlinear Programs. As these are very large classes of mathematical programs, the methods studied through the next chapters can be applied to an enormous number of practically relevant problems. Their importance is not limited to the specific problem of shortest paths; hence, we test our ideas on large sets of benchmark instances taken from the literature, in order to compare with other works and assess the usefulness of our approaches.

The rest of this part is organized as follows. In Chapter 7, we propose a modification in the branching rules of Branch-and-Bound algorithms for Mixed-Integer Linear Programs. We present two methods to generate good disjunctions of the feasible set, and report computational experiments to show that branching on these disjunctions yields better results with respect to traditional branching rules. In Chapter 8 we present an effective heuristic for Mixed-Integer Nonlinear Programs. Our heuristic is based on Variable Neighbourhood Search, and puts together several ingredients to create a very fast method that shows good performance on a large collection of test problems. Finally, in Chapter 9 we report computational results on solving time-dependent shortest paths instances with the methods that we discuss in this chapter.

Chapter 7

Improved Strategies for Branching on General Disjunctions

Mixed Integer Linear Programs (MILPs) arise in several real-life applications; in Section 1.3.2 we presented a mathematical program which models the time-dependent shortest path problem on generic (i.e. non-FIFO) networks with linear or piecewise linear arc cost functions. Typically, MILPs are solved via a Branch-and-Bound (88) or Branch-and-Cut algorithm such as that implemented by Cplex (83) or CBC (33), where the node bound is obtained by solving a Linear Programming (LP) relaxation of the MILP. Usually, branching occurs on the domain of integer variables in the form $x_j \leq k$ on one branch and $x_j \geq k+1$ on the other branch, where k is an integer. However, this need not be so: any disjunction not excluding points that are feasible in the original MILP can be used for branching. We use the term *branching on general disjunctions* to mean a branching strategy where the disjunctions are two disjoint halfspaces of the form $\pi x \leq \beta_0, \pi x \geq \beta_1$ with $\beta_0 < \beta_1$. Branching on a general disjunction is considered impractical because of the large computational effort needed to find a suitable branching direction π. A recent paper (84) proposes branching on general disjunctions arising from Gomory Mixed-Integer Cuts (GMICs). GMICs can be viewed as intersection cuts (10). At each node there is a choice of possible GMICs from which to derive the branching disjunction. The branching strategy suggested in (84) is based on the distance cut off by the corresponding intersection cut as a quality measure for the choice of disjunction. The improvement in objective function value that occurs after branching is at least as large as the improvement obtained after adding the corresponding intersection cut. In this chapter, we propose a modification in the class of disjunctions used for branching; instead of simply computing the disjunctions that define GMICs at the optimal basis, we try to generate a new set of disjunctions in order to increase the distance cut off by the corresponding intersection cut. By combining branching on simple disjunctions and on general disjunctions we obtain an improvement over traditional branching rules on the majority of test instances. More-

over, we propose a mathematical program that models the problem of finding a split disjunction that closes a large integrality gap as a MILP. At each node of the enumeration tree we solve this problem through heuristics and a local branching strategy in order to obtain several feasible solutions that represent split disjunctions. We apply strong branching to select one disjunction from this pool. Computational experiments show that on several instances of our test set we are able to reduce the number of required nodes by a large factor, although in terms of computational times, this approach is not competitive with the quadratic approach.

The rest of this section is organized as follows. In Section 7.1 we introduce our notation and the preliminaries which are necessary for the following. In Section 7.2 we propose a heuristic procedure, based on a quadratic formulation, that tries to generate good disjunctions by computing combinations of the rows of the simplex tableau. In Section 7.3 we give a mathematical formulation that models the problem of finding a split disjunction closing a large gap, and we discuss a local search algorithm that starts with a set of violated disjunctions and attempts to enlarge this set. In Section 7.4 we provide extensive computational experiments to test the heuristic approach, then we propose a combination of branching on general disjunctions and on single variables, and test the effectiveness in practice of this approach. In Section 7.5 we discuss the implementation of the mathematical formulation that models the problem of finding a split disjunction closing a large gap, and give computational results.

7.1 Preliminaries and notation

In this chapter we consider the Mixed Integer Linear Program in standard form:

$$\left.\begin{array}{rrcl} \min & c^\top x & & \\ & Ax & = & b \\ & x & \geq & 0 \\ \forall j \in N_I & x_j & \in & \mathbb{Z}, \end{array}\right\} \mathcal{P} \tag{7.1}$$

where $c \in \mathbb{R}^n$, $b \in \mathbb{R}^m$, $A \in \mathbb{R}^{m \times n}$ and $N_I \subset N = \{1, \ldots, n\}$. The LP relaxation of (7.1) is the linear program obtained by dropping the integrality constraints, and is denoted by $\bar{\mathcal{P}}$. The Branch-and-Bound algorithm makes an implicit use of the concept of disjunctions (11): whenever the solution of the current relaxation is fractional, we divide the current problem $\mathcal{P}$ into two subproblems $\mathcal{P}_1$ and $\mathcal{P}_2$ such that the union of the feasible regions of $\mathcal{P}_1$ and $\mathcal{P}_2$ contains all feasible solutions to $\mathcal{P}$. Usually, this is done by choosing a fractional component $\bar{x}_i$ (for some $i \in N_I$) of the optimal solution $\bar{x}$ to the relaxation $\bar{\mathcal{P}}$, and adding the constraints $x_i \leq \lfloor \bar{x}_i \rfloor$ and $x_i \geq \lceil \bar{x}_i \rceil$ to $\mathcal{P}_1$ and $\mathcal{P}_2$ respectively.

Within this chapter, we take the more general approach whereby branching can occur with respect to a direction $\pi \in \mathbb{R}^n$ by adding the constraints $\pi x \leq \beta_0$,

$\pi x \geq \beta_1$ with $\beta_0 < \beta_1$ to $\mathcal{P}_1$ and $\mathcal{P}_2$ respectively, as long as no integer feasible point is cut off. Owen and Mehrotra (118) generated branching directions π where $\pi_j \in \{-1, 0, +1\}$ for all $j \in N_I$ and showed that using such branching directions can decrease the size of the enumeration tree significantly. Aardal et al. (1) used basis reduction to find good branching directions for certain classes of difficult integer programs. Karamanov and Cornuéjols (84) proposed using disjunctions arising from GMICs generated directly from the rows of the optimal tableau. Given $B \subset N$ an optimal basis of $\bar{\mathcal{P}}$, and $J = N \setminus B$, i.e. J is the set of nonbasic variables, the corresponding simplex tableau is given by

$$x_i = \bar{x}_i - \sum_{j \in J} \bar{a}_{ij} x_j \quad \forall i \in B. \tag{7.2}$$

For $j \in J$, let $r^j \in \mathbb{R}^n$ be defined as

$$r_i^j = \begin{cases} -\bar{a}_{ij} & \text{if } i \in B \\ 1 & \text{if } i = j \\ 0 & \text{otherwise.} \end{cases} \tag{7.3}$$

These vectors are the extreme rays of the cone $\{x \in \mathbb{R}^n \mid Ax = b \wedge \forall j \in J \ (x_j \geq 0)\}$ with apex $\bar{x}$. Let $D(\pi, \pi_0)$ define the *split disjunction* $\pi^\top x \leq \pi_0 \vee \pi^\top x \geq \pi_0 + 1$, where $\pi \in \mathbb{Z}^n, \pi_0 \in \mathbb{Z}, \pi_j = 0$ for $i \notin N_I, \pi_0 = \lfloor \pi^\top \bar{x} \rfloor$. By integrality of (π, π_0), any feasible solution of $\mathcal{P}$ satisfies every split disjunction. Let $\epsilon(\pi, \pi_0) = \pi^\top \bar{x} - \pi_0$ be the violation by $\bar{x}$ of the first term of $D(\pi, \pi_0)$. Assume that the disjunction $D(\pi, \pi_0)$ is violated by $\bar{x}$, i.e. $0 < \epsilon(\pi, \pi_0) < 1$. Balas (10) defines the *intersection cut* associated with a basis B and a split disjunction $D(\pi, \pi_0)$ as

$$\sum_{j \in J} \frac{x_j}{\alpha_j(\pi, \pi_0)} \geq 1, \tag{7.4}$$

where for all $j \in J$ we define

$$\alpha_j(\pi, \pi_0) = \begin{cases} -\frac{\epsilon(\pi, \pi_0)}{\pi^T r^j} & \text{if } \pi^T r^j < 0 \\ \frac{1 - \epsilon(\pi, \pi_0)}{\pi^T r^j} & \text{if } \pi^T r^j > 0 \\ +\infty & \text{otherwise.} \end{cases} \tag{7.5}$$

The Euclidean distance between $\bar{x}$ and the cut defined above is (see (12)):

$$\delta(B, \pi, \pi_0) = \sqrt{\frac{1}{\sum_{j \in J} \frac{1}{\alpha_j(\pi, \pi_0)^2}}}. \tag{7.6}$$

7.2 A quadratic optimization approach

Assume that the optimal solution $\bar{x}$ to the LP relaxation $\bar{\mathcal{P}}$ is not feasible to $\mathcal{P}$. We would like to generate a good branching disjunction $D(\pi, \pi_0)$. In (84) it is

shown that the gap closed by branching on a disjunction $D(\pi, \pi_0)$ is at least as large as the improvement in the objective function obtained by the corresponding intersection cut. Thus, it makes sense to attempt to increase the value of the distance $\delta(B, \pi, \pi_0)$ as much as possible. It is easy to see from (7.6) that this means increasing the value of $\alpha_j(\pi, \pi_0)$ for all $j \in J$, which in turn corresponds to decreasing the coefficient of the intersection cut (7.4). (84) considered intersection cuts (7.4) obtained directly from the optimal tableau (7.2) as GMICs for $i \in B \cap N_I$ such that $\bar{x}_i \notin \mathbb{Z}$.

In this section we consider split disjunctions arising from GMICs generated from linear combinations of the rows of the simplex tableau (7.2):

$$\sum_{i \in B} \lambda_i x_i = \tilde{x} - \sum_{j \in J} \tilde{a}_j x_j, \tag{7.7}$$

where

$$\begin{array}{rcl} \tilde{x} & = & \sum_{i \in B} \lambda_i \bar{x}_i \\ \tilde{a}_j & = & \sum_{i \in B} \lambda_i \bar{a}_{ij} \text{ for } j \in J. \end{array} \tag{7.8}$$

Note that in order to generate a GMIC we need $\tilde{x} \notin \mathbb{Z}$ and therefore $\lambda \neq 0$. For $i \in B$, it will be convenient to define $\tilde{a}_i = \lambda_i$. The GMIC associated with (7.7) is the inequality

$$\sum_{j \in N} \frac{x_j}{\alpha_j} \geq 1 \tag{7.9}$$

where

$$\alpha_j = \begin{cases} \max\left(\frac{\tilde{x} - \lfloor \tilde{x} \rfloor}{\tilde{a}_j - \lfloor \tilde{a}_j \rfloor}, \frac{\lceil \tilde{x} \rceil - \tilde{x}}{\lceil \tilde{a}_j \rceil - \tilde{a}_j}\right) & \text{if } j \in N_I \\ \max\left(\frac{\tilde{x} - \lfloor \tilde{x} \rfloor}{\tilde{a}_j}, \frac{\lceil \tilde{x} \rceil - \tilde{x}}{-\tilde{a}_j}\right) & \text{if } j \in N \setminus N_I. \end{cases} \tag{7.10}$$

By convention, α_j is equal to $+\infty$ when one of the denominators is zero in (7.10). Note that the GMIC (7.9) may not cut off $\bar{x}$ when λ_i is not integer for $i \in B \cap N_I$ or $\lambda_i \neq 0$ for $i \in B \setminus N_I$. On the other hand, when λ is integral for $i \in B \cap N_I$ and $\lambda_i = 0$ for $i \in B \setminus N_I$ the GMIC (7.9) is of the form (7.4) since all the basic variables have a zero coefficient $1/\alpha_j$. In this case, the distance cut off is given by (7.6). For this reason, we restrict ourselves to nonzero integral multipliers λ. In this case, the split disjunction $D(\pi, \pi_0)$ that defines the GMIC associated to (7.7) can be computed as (see (10; 72)):

$$\begin{array}{rcl} \pi_j & = & \begin{cases} \lfloor \tilde{a}_j \rfloor & \text{if } j \in N_I \cap J \text{ and } \tilde{a}_j - \lfloor \tilde{a}_j \rfloor \leq \tilde{x}_i - \lfloor \tilde{x} \rfloor \\ \lceil \tilde{a}_j \rceil & \text{if } j \in N_I \cap J \text{ and } \tilde{a}_j - \lfloor \tilde{a}_j \rfloor > \tilde{x} - \lfloor \tilde{x} \rfloor \\ \lambda_j & \text{if } j \in N_I \cap B \\ 0 & \text{otherwise}, \end{cases} \\ \pi_0 & = & \left\lfloor \pi^\top \tilde{x} \right\rfloor. \end{array} \tag{7.11}$$

It is easy to check that if we plug (7.11) into $\alpha_j(\pi, \pi_0)$ as defined in (7.5) we get exactly α_j as defined in (7.10) for $j \in J$. Since $1/\alpha_j = 0$ for $j \in B$, this shows that the GMIC (7.9) is an intersection cut.

In this section we study a method for decreasing $1/\alpha_j$ for $j \in J$. By (7.6), this yields a disjunction with a larger value of $\delta(B, \pi, \pi_0)$, which is thus likely to close a larger gap. To achieve this goal, we choose an integral vector λ that defines (7.7), which we have seen to have an influence on both the numerators and the denominators of (7.10) through (7.8). It seems difficult to optimize α_j for $j \in J \cap N_I$ because both terms of the fraction are nonlinear. Furthermore, for $j \in N_I$, $1/\alpha_j$ is always between 0 and 1 independent of the choice of λ. For $j \in J \setminus N_I$ the coefficient $1/\alpha_j$ is not bounded, therefore we concentrate on these coefficients. From (7.10) we see that the denominator of α_j for $j \in J \setminus N_I$ is a linear function of λ through (7.8), whereas the numerator is a nonlinear function of λ and is always between 0 and 1. For this reason we attempt to minimize $\tilde{a}_j$ for $j \in J \setminus N_I$ over integral vectors λ. More specifically, we would like to minimize $\|\tilde{d}\|$, where

$$\tilde{d} = (\tilde{a}_j)_{j \in J \setminus N_I} . \tag{7.12}$$

Let $B_I = B \cap N_I$, $J_C = J \setminus N_I$; apply a permutation to the simplex tableau in order to obtain $B_I = \{1, \ldots, |B_I|\}, J_C = \{1, \ldots, |J_C|\}$, and define the matrix $D \in \mathbb{R}^{|B_I| \times |J_C|}$, $d_{ij} = \bar{a}_{ij}$. Minimizing $\|\tilde{d}\|$ can be written as

$$\min_{\lambda \in \mathbb{Z}^{|B_I|} \setminus \{0\}} \| \sum_{i \in B_I} \lambda_i d_i \|. \tag{7.13}$$

This is a shortest vector problem in the additive group generated by the rows of D. If these rows are linearly independent, the group defines a lattice, and we have the classical shortest vector problem in a lattice, which is NP-hard under randomized reductions (6).

Andersen, Cornuéjols and Li (8) proposed a heuristic for (7.13) based on a reduction algorithm which cycles through the rows of D and, for each such row d_k, considers whether summing an integer multiple of some other row yields a reduction of $\|d_k\|$. If this is the case, the matrix D is updated by replacing d_k with the shorter vector. Note, however, that this method only considers two rows at a time.

Our idea is to use, for each row d_k of D, a subset $R_k \subset B_I$ of the rows of the simplex tableau with $d_k \in R_k$, in order to reduce $\|d_k\|$ as much as possible with a linear combination with integer coefficients of d_k and d_i for all $i \in R_k \setminus \{k\}$. This is done by defining, for each row d_k that we want to reduce, the convex minimization problem:

$$\min_{\lambda^k \in \mathbb{R}^{|R_k|}, \lambda^k_k = 1} \| \sum_{i \in R_k} \lambda^k_i d_i \|, \tag{7.14}$$

and then rounding the coefficients λ^k_i to the nearest integer $\lfloor \lambda^k_i \rceil$. There are several reasons for imposing $\lambda^k_k = 1$. One reason is that not only do we want to solve the shortest vector problem, but it is also important to find a vector λ^k with small norm. We will come back to this issue in Section 7.2.1. Another

reason is that we must avoid the zero vector as a solution. Yet another is to get different optimization problems for $k = 1, \ldots, |B_I|$, thus increasing the chance of obtaining different branching directions. Vanishing the partial derivatives of $\| \sum_{i \in R_k} \lambda_i^k d_i \|$ with respect to λ_i^k for all i, we obtain an $|R_k| \times |R_k|$ linear system that yields the optimal (continuous) solution. We formalize our problem: for $k = 1, \ldots, |B_I|$ we solve the linear system

$$A^k \lambda^k = b^k,$$

where $A^k \in \mathbb{R}^{|R_k| \times |R_k|}$ and $b^k \in \mathbb{R}^{|R_k|}$ are defined as follows:

$$\begin{aligned} A_{ij}^k &= \begin{cases} 1 & \text{if } i = j = k \\ 0 & \text{if } i = k \text{ or } j = k \text{ but not both} \\ \sum_{h=1}^{|J^C|} d_{ih} d_{jh} & \text{otherwise,} \end{cases} \\ b_i^k &= \begin{cases} 1 & \text{if } i = k \\ -\sum_{h=1}^{|J^C|} d_{ih} d_{kh} & \text{otherwise.} \end{cases} \end{aligned} \tag{7.15}$$

The form of the linear system guarantees $\lambda_k^k = 1$ in the solution.

Once these linear systems are solved and we have the optimal coefficients $\lambda^k \in \mathbb{R}^{|R_k|}$ for all $k \in \{1, \ldots, |B_I|\}$, we round them to the nearest integer. Then, we consider the norm of $\sum_{i \in R_k} \lfloor \lambda_i^k \rceil d_i$. If $\| \sum_{i \in R_k} \lfloor \lambda_i^k \rceil d_i \| < \| d_k \|$, then we have an improvement with respect to the original row of the simplex tableau; in this case, we use

$$\sum_{i \in R_k} \lambda_i^k x_i = \sum_{i \in R_k} \lambda_i^k \bar{x}_i - \sum_{j \in J} \sum i \in R_k \lambda_i^k \bar{a}_{ij} x_j, \tag{7.16}$$

instead of row $\bar{a}_k$ in order to compute a GMIC, and consider the possibly improved disjunction for branching.

Example 7.2.1. *Suppose we have the following matrix D:*

$$D = \begin{bmatrix} 3 & 1 & 8 & 2 & 3 & 2 & 3 \\ 1 & -2 & 0 & 12 & -2 & -4 & -5 \\ 0 & -1 & 4 & 1 & 4 & 5 & -1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 2 \end{bmatrix}$$

and we apply the reduction algorithm to the first row d_1. Following (7.15), we obtain the linear system:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 194 & -9 & 1 \\ 0 & -9 & 60 & 2 \\ 0 & 1 & 2 & 8 \end{bmatrix} \begin{bmatrix} \lambda_1^1 \\ \lambda_2^1 \\ \lambda_3^1 \\ \lambda_4^1 \end{bmatrix} \begin{bmatrix} 1 \\ 4 \\ -52 \\ -20 \end{bmatrix},$$

whose solution is $\lambda^1 = (\lambda^1_1, \lambda^1_2, \lambda^1_3, \lambda^1_4)^\top = (1, -0.0042, -0.7906, -2.3018)^\top$. *Rounding each component to the nearest integer and computing* $\lfloor \lambda^1 \rceil^\top D$ *we obtain the row:*

$$\begin{bmatrix} 1 & 0 & 2 & -1 & -1 & -3 & 0 \end{bmatrix},$$

whose L_2 *norm is 4, as opposed to the initial norm of* d_1*, which is 10. Thus, we compute the corresponding row of the simplex tableau with the same coefficients* λ^1*, and consider the possibly improved disjunction for branching.*

On the other hand, if we apply the reduction algorithm to the second row of D*, we obtain the linear system:*

$$\begin{bmatrix} 100 & 0 & 52 & 20 \\ 0 & 1 & 0 & 1 \\ 52 & 0 & 60 & 2 \\ 20 & 0 & 2 & 8 \end{bmatrix} \begin{bmatrix} \lambda^2_1 \\ \lambda^2_2 \\ \lambda^2_3 \\ \lambda^2_4 \end{bmatrix} = \begin{bmatrix} 4 \\ 1 \\ 9 \\ -1 \end{bmatrix},$$

whose solution is $\lambda^2 = (\lambda^2_1, \lambda^2_2, \lambda^2_3, \lambda^2_4)^\top = (-0.0627, 1, 0.2050, -0.0196)^\top$. *Rounding each component to the nearest integer and computing* $\lfloor \lambda^2 \rceil^\top D$ *gives the original row* d_2*, hence the reduction algorithm did not yield an improvement.*

7.2.1 The importance of the norm of λ

Although solving the shortest vector problem (7.13) is important for finding a deep cut, it is not the only consideration when trying to find a good branching direction. In the space $B \cap N_I$, the distance between the two hyperplanes that define a split disjunction $D(\pi, \pi_0)$ is related to the norm of λ, and is in fact equal to $1/\|\lambda\|$ as can be seen from (7.11). Therefore, in this space disjunctions that cut off a larger volume will have a small $\|\lambda\|$. We illustrate this with an example.

Example 7.2.2. *Consider the following tableau, where* x_1, x_2 *are binary variables and* y_1, y_2 *are continuous:*

$$\begin{cases} x_1 &= 1/3 + 98y_1 + y_2 \\ x_2 &= 1/3 - 99y_1 - 1.01y_2 \end{cases} \tag{7.17}$$

The solution to the shortest vector problem (7.13) is given by the integer multipliers $\lambda_1 = 99, \lambda_2 = 98$ *which yield the shortest vector in the lattice* $(0, 0.02)$ *and the disjunction* $99x_1 + 98x_2 \leq 65 \vee 99x_1 + 98x_2 \geq 66$*. Our heuristic method computes the continuous multipliers* $\lambda_1 = 1, \lambda_2 = 98/99$ *which are rounded to* $\lambda_1 = 1, \lambda_2 = 1$*, that correspond to the disjunction* $x_1 + x_2 \leq 0 \vee x_1 + x_2 \geq 1$*. It easy to verify that the distance between these two hyperplanes is roughly ten times larger than in the first case. Therefore, in the unit square, the disjunction obtained through our heuristic method dominates the one computed through the exact solution of the shortest vector problem. Figure 7.1 gives a picture of this.*

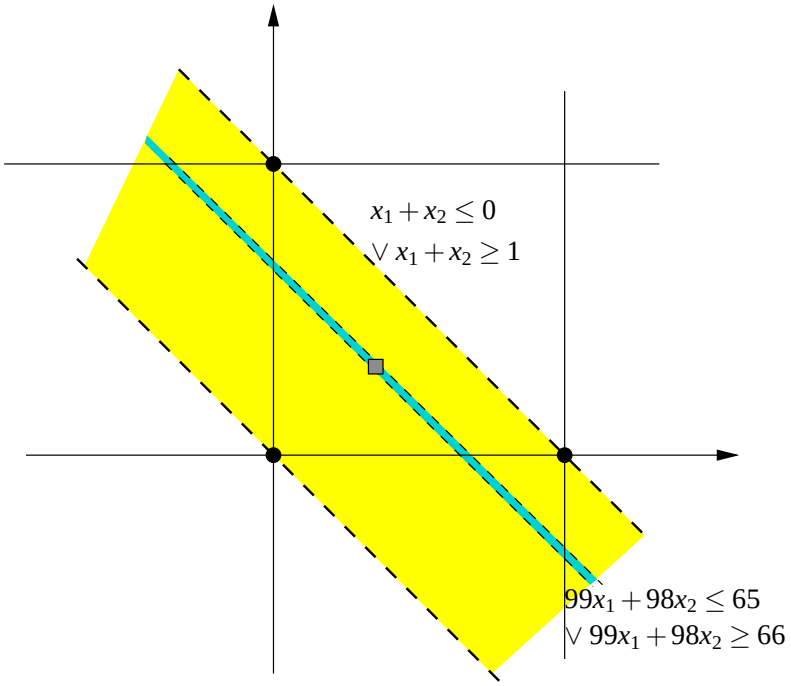


Figure 7.1: Representation of the disjunctions discussed in Example 7.2.2.

7.2.2 Choosing the set R_k

The choice of each set $R_k \subset B_I$ for all k has an effect on the performance of the norm reduction algorithm. Although using $R_k = B_I$ is possible, in that case two problems arise: first, the size of the linear systems may become unmanageable in practice, and second, if we add up too many rows then the coefficients on the variables with indices in $J \cap N_I$ may deteriorate. In particular, we may get more nonzero coefficients. Thus, we do the following. We fix a maximum cardinality $M_{|R_k|}$; if $M_{|R_k|} \geq |B_I|$, we set $R_k = B_I$. Otherwise, for each row k that we want to reduce, we sort the remaining rows by ascending number of nonzero coefficients on the variables with indices in $\{i \in J \cap N_I | \bar{a}_{ki} = 0\}$, and select the first $M_{|R_k|}$ indices as those in R_k. The reason for this choice is that, although the value of the coefficients on the variables with indices in $J \cap N_I$ is bounded, we would like those that are zero in row $\bar{a}_k$ to be left unmodified when we compute $\sum_{j \in R_k} \lfloor \lambda_j^k \rceil \bar{a}_j$. As zero coefficients on those variables yield a stronger cut, we argue that this will yield a stronger split disjunction as well. During the sorting operation, to keep computational times to a minimum we use the row number as a tie breaker.

7.2.3 The depth of the cut is not always a good measure

As stated earlier, the improvement in the objective function given by a GMIC is a lower bound on the improvement obtained by branching on the corresponding disjunction. We give an example showing that this lower bound may not be tight and in fact the difference between these two values can be arbitrarily large.

Example 7.2.3. *Consider the integer program:*

$$\left.\begin{array}{rrcl} \min & -x_1 - x_2 & & \\ & x_1 & \leq & 1.5 \\ & x_2 & \leq & 1 \\ & x_1/m - x_2 & \geq & 1.5/m - 1.25 \\ & mx_1 - x_2 & \leq & 1.5m - 0.75 \\ x_1, x_2 & & \in & \mathbb{Z}, \end{array}\right\} \mathcal{P} \tag{7.18}$$

where $m > 1$ is a given parameter close to 1. The solution to the LP relaxation is $(1.5, 1)$, with an objective value of -2.5. The intersection cut obtained from the disjunction $x_1 - x_2 \leq 0 \vee x_1 - x_2 \geq 1$ is $x_1 + x_2 \leq 2$, which gives an objective value of -2. Now suppose we branch on $x_1 - x_2 \leq 0 \vee x_1 - x_2 \geq 1$. We obtain two children $\mathcal{P}_1$ and $\mathcal{P}_2$, which are both feasible. One can verify that optimal solution of $\mathcal{P}_1$ is $(\frac{1.5-1.25m}{1-m}, \frac{1.5-1.25m}{1-m})$ with objective value $-2\frac{1.5-1.25m}{1-m}$, and the optimal solution of $\mathcal{P}_2$ is $(\frac{1.5m-1.75}{m-1}, \frac{0.5m-0.75}{m-1})$ with objective value $-\frac{2m-2.5}{m-1}$. Therefore, the gap closed by branching is:

$$\max\{-2\frac{1.5 - 1.25m}{1 - m}, -\frac{2m - 2.5}{m - 1}\} - 2.5,$$

which can be made arbitrarily large when m tends to 1 from above. At the same time, the intersection cut associated with the same disjunction closes a gap of 0.5 regardless of m. We give a picture of the situation for $m = 1.1$ in Figure 7.2.

Example 7.2.3 suggests that we should not choose the branching direction by relying on the distance cut off by the intersection cut only, even though the quality of the underlying intersection cut gives an indication of the strength of a disjunction and can guide us towards generating better disjunctions. Therefore, in our computational experiments (see Section 7.4) we employed strong branching to select a disjunction among those computed with the procedure described so far.

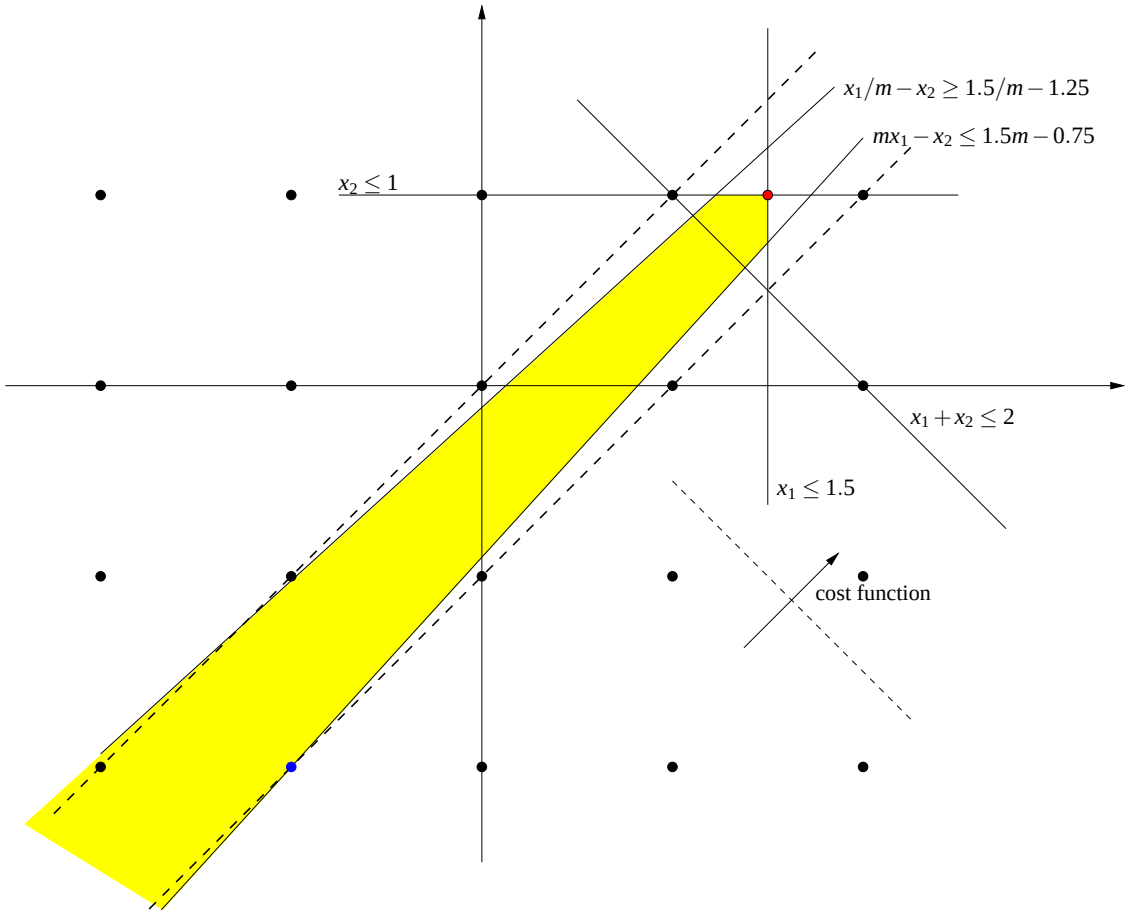


Figure 7.2: Representation of Example 7.2.3 for $m = 1.1$.

7.3 A MILP formulation to generate split disjunctions

For $j \in J$ let the intersection points between the ray r^j and a split disjunction $D(\pi, \pi_0)$ be:

$$x^j = \bar{x} + \alpha_j(\pi, \pi_0) r^j \tag{7.19}$$

Let $\mathcal{P}_1, \mathcal{P}_2$ be the problems at the children nodes defined by the split disjunction, and $\bar{x}^1$ and $\bar{x}^2$ be the corresponding linear relaxation optima (if $\mathcal{P}_k$ is infeasible then $\bar{x}^k = \infty$ for $k \in \{1, 2\}$).

Lemma 7.3.1. $\min_{j \in J} c^\top x^j \leq \min(c^\top \bar{x}^1, c^\top \bar{x}^2)$.

Proof. Let $F_1 = P \cap \{x \in \mathbb{R}^n : \pi^\top x \leq \pi_0\}$ and $F_2 = P \cap \{x \in \mathbb{R}^n : \pi^\top x \geq \pi_0 + 1\}$ be the feasible regions of, respectively, $\mathcal{P}_1$ and $\mathcal{P}_2$; we have $\bar{x}^1 = \arg\min\{c^\top x : c \in F_1\}$, $\bar{x}^2 = \arg\min\{c^\top x : c \in F_2\}$. Let $F_1^r = P(B) \cap \{x \in \mathbb{R}^n : \pi^\top x \leq \pi_0\}$

and $F_2^r = P(B) \cap \{x \in \mathbb{R}^n : \pi^\top x \geq \pi_0 + 1\}$. By the definition of the points $x^j \; \forall j \in J$, we have that $\min_{j \in J}\{c^\top x\} = \min\{c^\top x : x \in F_1^r \cup F_2^r\}$. Since $P \subseteq P(B)$, $\min\{c^\top x : x \in F_1^r \cup F_2^r\} \leq \min\{c^\top x : x \in F_1 \cup F_2\} = \min(c^\top \bar{x}^1, c^\top \bar{x}^2)$, which completes the proof. □

Therefore, in order to generate children nodes that have tight LP relaxations, it makes sense to try to maximize the lower bound given in Lemma 7.3.1. In other words, we want to maximize the gap closed optimizing over the set of split disjunctions. This problem can be formalized as follows.

$$\max_{\substack{(\pi,\pi_0) \in \mathbb{Z}^{n+1} \\ \pi_0 = \lfloor \pi^\top \bar{x} \rfloor \\ \pi_i = 0 \forall i \in N \setminus N_I}} \; \min_{j \in J} c^\top (\bar{x} + \alpha_j(\pi, \pi_0) r^j). \tag{7.20}$$

This is a nonconvex Mixed Integer Nonlinear Problem (MINLP) where the complicating constraints are $\pi_0 = \lfloor \pi^\top \bar{x} \rfloor$ as well as the definition of α_j (for all $j \in J$). We use the same simplification given in (8) and assume that α_j's numerator (see Eq. (7.5)) is fixed; for simplicity, we assume its value to be 1. Since it is the only term where π_0 appears, we can discard π_0 from the problem altogether. Furthermore, the constant $c^\top \bar{x}$ can also be discarded:

$$\max_{\substack{\pi \in \mathbb{Z}^n \\ \pi_i = 0 \forall i \in N \setminus N_I}} \; \min_{j \in J} \alpha_j(\pi, \pi_0) c^\top r^j. \tag{7.21}$$

Since $\pi^\top r^j = 0$ means that there is no intersection between the split disjunction and the ray r^j, we can safely assume that $\pi^\top r^j \neq 0$. For all $j \in J$, we introduce variables α_j and constraints

$$\alpha_j = \frac{1}{|\pi^\top r^j|}. \tag{7.22}$$

The "max min" objective function is easily reformulated to:

$$\left.\begin{array}{rrcl} \max_\pi & y & & \\ \forall j \in J & y & \leq & \alpha_j c^\top r^j \\ \forall j \in J & \alpha_j & = & \frac{1}{|\pi^\top r^j|} \\ \forall i \in N \setminus N_I & \pi_i & = & 0 \\ & \pi & \in & \mathbb{Z}^n. \end{array}\right\} \tag{7.23}$$

By (7.22), $\alpha_j > 0$ for all $j \in J$, so (7.23) can be rewritten as:

$$\left.\begin{array}{rrcl} \max_\pi & y & & \\ \forall j \in J & \frac{y}{\alpha_j} & \leq & c^\top r^j \\ \forall j \in J & |\pi^\top r^j| & = & \frac{1}{\alpha_j} \\ \forall i \in N \setminus N_I & \pi_i & = & 0 \\ & \pi & \in & \mathbb{Z}^n. \end{array}\right\}$$

Remark 7.3.2. *Shifting $|\pi^\top r^j|$ from the denominator to a numerator implicitly removes the constraint $\pi^\top r^j \neq 0$.*

Next, we introduce variables w_j to replace the expression $\frac{1}{\alpha_j}$ for all $j \in J$, obtaining:

$$\left.\begin{array}{rrcl} \max_\pi & y & & \\ \forall j \in J & w_j y & \leq & c^\top r^j \\ \forall j \in J & |\pi^\top r^j| & = & w_j \\ \forall i \in N \setminus N_I & \pi_i & = & 0 \\ & \pi & \in & \mathbb{Z}^n. \end{array}\right\}$$

We reformulate the absolute value as follows:

$$\left.\begin{array}{rrcl} \max_\pi & y & & \\ \forall j \in J & w_j y & \leq & c^\top r^j \\ \forall j \in J & w_j & \geq & -\pi^\top r^j \\ \forall j \in J & w_j & \geq & \pi^\top r^j \\ \forall i \in N \setminus N_I & \pi_i & = & 0 \\ & \pi & \in & \mathbb{Z}^n. \end{array}\right\} \tag{7.24}$$

This is a bilinear problem of a special kind, namely one of the sets of variables involved in the products only contains the single decision variable y. It is easy to see that $w_j > 0$ for all j (it follows from the assumption $\pi^\top r^j \neq 0$).

Lemma 7.3.3. *For any y feasible in (7.24) we have $y \geq 0$.*

Proof. By the optimality conditions on $\bar{x}$, $c^\top r^j \geq 0$ (i.e. the optimization direction $-c$ makes an non-acute angle with each of the rays r^j), hence any y feasible in (7.24) is such that $y \geq 0$. □

The case $y = 0$ being uninteresting (there is no guarantee of any closed gap), we restrict our attention to $y > 0$. This allows us to divide the constraints $w_j y \leq c^\top r^j$ through by y, obtaining:

$$\forall j \in J \; w_j \leq \frac{1}{y} c^\top r^j. \tag{7.25}$$

We now introduce a positive variable $z = \frac{1}{y}$ to linearize (7.25):

$$\forall j \in J \; w_j \leq z c^\top r^j. \tag{7.26}$$

This transforms the objective function in $\max \frac{1}{z}$; since this is a monotonically decreasing univariate function, it can be reformulated as $\min z$. The problem

now becomes a MILP:

$$\left.\begin{array}{rrcl} \min & z & & \\ \forall j \in J & w_j & \leq & zc^\top r^j \\ \forall j \in J & w_j & \geq & -\pi^\top r^j \\ \forall j \in J & w_j & \geq & \pi^\top r^j \\ \forall i \in N \setminus N_I & \pi_i & = & 0 \\ & \pi & \in & \mathbb{Z}^n \\ & z & \geq & 0. \end{array}\right\} \tag{7.27}$$

By Remark 7.3.2, $\pi = w = 0$ and $z = 0$ is a feasible solution of (7.27) that is however infeasible in the original problem (7.20). We also have to make sure that the solution of (7.27) is a violated disjunction, i.e. $\pi^\top \bar{x}$ is fractional. To this end, we add another integer variable π_0, and impose the constraints that $\pi^\top \bar{x} - \pi_0 \geq \eta$ and $\pi_0 + 1 - \pi^\top \bar{x} \geq \eta$, where η is a small positive tolerance. As for each vector (π, π_0) the symmetric one $(-\pi, -\pi_0 - 1)$ represents the same disjunction, we eliminate some symmetric solutions of the problem by imposing $\pi_0 \geq 0$. We obtain:

$$\left.\begin{array}{rrcl} \min & z & & \\ \forall j \in J & w_j & \leq & zc^\top r^j \\ \forall j \in J & w_j & \geq & -\pi^\top r^j \\ \forall j \in J & w_j & \geq & \pi^\top r^j \\ \forall i \in N \setminus N_I & \pi_i & = & 0 \\ & \pi^\top \bar{x} - \pi_0 & \geq & \eta \\ & \pi_0 + 1 - \pi^\top \bar{x} & \geq & \eta \\ & (\pi, \pi_0) & \in & \mathbb{Z}^{n+1} \\ & \pi_0 & \geq & 0 \\ & z & \geq & 0. \end{array}\right\} \tag{7.28}$$

In order to reduce the size of the search space and speed up the solution process, we add a constraint on the 1-norm of π, such as $\sum_{i \in N} |\pi_i| \leq K$. This constraint can be reformulated in linear form by introducing two positive variables

π_i^+ and π_i^- such that $\pi_i = \pi_i^+ - \pi_i^- \; \forall i \in N$. Finally (7.28) becomes:

$$\left.\begin{array}{rrcl} \min & z & & \\ \forall j \in J & w_j & \leq & zc^\top r^j \\ \forall j \in J & w_j & \geq & -\pi^\top r^j \\ \forall j \in J & w_j & \geq & \pi^\top r^j \\ \forall i \in N \setminus N_I & \pi_i & = & 0 \\ \forall i \in N & \pi_i^+ - \pi_i^- & = & \pi_i \\ & \sum_{i \in N}(\pi_i^+ + \pi_i^-) & \leq & K \\ & \pi^\top \bar{x} - \pi_0 & \geq & \eta \\ & \pi_0 + 1 - \pi^\top \bar{x} & \geq & \eta \\ & (\pi, \pi_0) & \in & \mathbb{Z}^{n+1} \\ & (\pi^+, \pi^-) & \in & \mathbb{Z}_+^{2n} \\ & \pi_0 & \geq & 0 \\ & z & \geq & 0. \end{array}\right\} \tag{7.29}$$

7.3.1 Generating a pool of split disjunctions

The MILP (7.29) is a linear approximation of the original problem (7.20), thus solving it to optimality does not guarantee to give the disjunction which maximizes the lower bound in Lemma 7.3.1. Besides, in Section 7.2.3 we provided an example to show that we should not rely on the distance cut off only to evaluate the strength of a disjunction. Therefore, we are interested in finding several feasible solutions to (7.29), so as to have a large pool of split disjunctions which can be evaluated via strong branching to obtain the closed gap:

$$\min(c^\top \bar{x}^1, c^\top \bar{x}^2) - c^\top \bar{x}.$$

Some feasible solutions are typically found during the exploration of the branch-and-bound tree while solving (7.29). We repeatedly employ a local branching approach (56) from different starting points in order to increase the number of feasible solutions to (7.29) computed in a short time. We now endeavour to explain in more detail this approach.

Local branching was proposed by Fischetti and Lodi (56) as an effective heuristic for MILPs. The original idea only takes into account 0-1 variables, but the authors give an extension to case of general integer variables. Local branching consists in exploring the neighbourhood of a starting feasible solution by adding a local branching constraing, which limits the number of binary variables allowed to change their value with respect to the starting solution to at most a given parameter K. K is then the size of the neighbourhood. If the value of K is small, the neighbourhood is rapidly explored by MILP solvers, often leading to a better solution then the original one, as shown by the computational experiments in (56).

We are interested in using a local branching paradigm as a mean to rapidly obtain more feasible solutions to (7.29), possibly also improving the objective function value. Suppose we already have a branching direction $\bar{\pi} \in \mathbb{Z}^n$ as starting solution. The space of vectors $\pi \in \mathbb{Z}^n$ such that π is obtained from $\bar{\pi}$ by adding or subtracting 1 to one component can be explored by adding to (7.29) a constraint that forces at least one variable π_i^+ or π_i^- to increase by 1. We call this space the *+/-1 neighbourhood* of $\bar{\pi}$. Note that it is possible to obtain the initial disjunction if both π_i^+ and π_i^- for a given $i \in N$ increase by one, therefore we exclude this case by allowing exactly one variable to increase. The resulting problem is:

$$\left.\begin{array}{rrcl}
\min & z & & \\
\forall j \in J & w_j & \leq & z c^\top r^j \\
\forall j \in J & w_j & \geq & -\pi^\top r^j \\
\forall j \in J & w_j & \geq & \pi^\top r^j \\
\forall i \in N \setminus N_I & \pi_i & = & 0 \\
\forall i \in N & \pi_i^+ - \pi_i^- & = & \pi_i \\
\forall i \in N & \bar{\pi}_i^+ \leq \pi_i^+ & \leq & \bar{\pi}_i^+ + 1 \\
\forall i \in N & \bar{\pi}_i^- \leq \pi_i^- & \leq & \bar{\pi}_i^- + 1 \\
& \sum_{i \in N} (\pi_i^+ - \bar{\pi}_i^+ + \pi_i^- - \bar{\pi}_i^-) & = & 1 \\
& \pi^\top \bar{x} - \pi_0 & \geq & \eta \\
& \pi_0 + 1 - \pi^\top \bar{x} & \geq & \eta \\
& (\pi, \pi_0) & \in & \mathbb{Z}^{n+1} \\
& (\pi^+, \pi^-) & \in & \mathbb{Z}_+^{2n} \\
& \pi_0 & \geq & 0 \\
& z & \geq & 0.
\end{array}\right\} \tag{7.30}$$

We repeatedly solve problem (7.30), each time with a different starting vector $\bar{\pi}$. Let $obj(\pi)$ be the value of the objective function of the problem (7.29) associated with a feasible solution π. We apply the following neighbourhood search algorithm. The algorithm tries to generate new feasible solutions to (7.29) and terminates only when there are at least M disjunctions in the disjunction pool or there are no more solution neighbourhoods to explore. As the solution space of each problem (7.30) solved by Algorithm 4 is small, its application requires a very small computational time if compared to the problem without local branching constraints (7.29). At each iteration of the neighbourhood search algorithm, we avoid finding solutions already included in the current set of split disjunctions D' by fixing some variables. In particular, when solving (7.30) with a given starting point $\bar{\pi}$ we check whether there is one (or more) vector $\pi' \in D'$ in the $+/-1$ neighbourhood of $\bar{\pi}$, and, if this is the case, we fix the variable π_i^+ or π_i^- that transforms $\bar{\pi}$ into π', so that π' cannot be obtained from (7.30).

Algorithm 4 The neighbourhood search algorithm.

INPUT: A set D of split disjunctions;
maximum number of split disjunctions M.
OUTPUT: A set D' of split disjunctions such that $|D'| \geq |D|$.
Set $D' \leftarrow D$, s$top \leftarrow$ `false`
while $|D'| \leq M \wedge \neg$stop **do**
  if $\exists \pi \in D' : \pi$ never used as starting solution **then**
    Set $\bar{\pi} \leftarrow \arg\min\{obj(\pi) | \pi \in D' \wedge \pi$ never used as starting solution $\}$
  else
    Set s$top \leftarrow$ `true`
  if $\neg$stop **then**
    Solve problem (7.30) with $\bar{\pi}$ as starting solution
    Add all new solutions to D'

7.4 Computational experiments: quadratic approach

To assess the usefulness of our approach, we implemented within the Cplex 11.0 Branch-and-Bound framework the following branching methods:

- branching on single variables (Simple Disjunctions, SD),
- branching on split disjunctions after the reduction step that we proposed in Section 7.2 (Improved General Disjunctions, IGD),
- branching on the split disjunctions that define the GMICs at the current basis (General Disjunctions, GD), in order to compare with the work of Karamanov and Cornuéjols (84),
- branching on split disjunctions after the application of the Reduce-and-Split reduction algorithm (Reduce-and-Split, RS), in order to compare with the work of Andersen, Cornuéjols and Li (8),
- a combination of the SD and IGD method (Combined General Disjunction, CGD), which is described in Section 7.4.2.

In each set of experiments we applied only the methods that were meaningful for that particular experiment. We applied strong branching in order to choose the best branching decision. When not otherwise stated, the best branching decision is considered to be the one that generates the smallest number of feasible children, or, in the case of a tie, the one that closes more gap, computed as $\min\{c^\top \bar{x}^1, c^\top \bar{x}^2\}$ where $\bar{x}^1, \bar{x}^2$ are the optimal solutions of the LP relaxations of the children nodes. If a branching decision generates only one feasible child at the current node, one side of the disjunction (i.e. the feasible one) can be

considered as a cutting plane; when several disjunctions of this kind are discovered, we add all these cutting planes. This leads to only one feasible child, but with possibly a larger closed gap with respect to the case where we add only one disjunction as branching constraint. Unless otherwise stated, our testbed is the union of `miplib2.0`, `miplib3` and `miplib2003`, after the removal of all instances that can be solved to optimality in less than 10 nodes by all algorithms[1], and the instances where one node cannot be processed in less than 30 minutes (including strong branching) by the SD algorithm[2]. In total, the set consists in 96 heterogeneous instances. The node selection strategy was set to *best bound*, and the value of the optimal solution was given as a cutoff value for all those instances where the optimum is known[3]. These choices were meant to reduce as much as possible the size of the enumeration tree, and to minimize the effect of heuristics and of other uncontrollable factors (such as the time to find the first integer solution) in order to get a more stable indication of the algorithm performance on branching.

The rest of this section is divided as follows. In Section 7.4.1 we consider the different branching algorithms separately, and compare them in several respects. In Section 7.4.2 we capitalize on the insight gained with the different experiments of the previous section, and we combine the methods into a single branching algorithm, which we test in a Branch-and-Cut framework. All averages reported in the following are geometric averages; to compute the geometric average of a set of values not necessarily greater than zero, we summed 1 to all values before computing the mean, and subtracted 1 from the result.

7.4.1 Comparison of the different methods

The first set of experiments involves branching at root node in order to evaluate the amount of integrality gap closed; we compute the relative closed integrality gap as $\frac{\text{closed gap}}{\text{initial gap}}$ for those instances where the optimal solution is known, while for other instances we simply compare the absolute closed gap. In this set of experiments we evaluated all possible branching decisions via strong branching: that is, for SD we branched on all fractional integer variables, for GD we branched on the split disjunctions computed from the rows of the simplex tableau where the associated basic variable is a fractional integer variable, and for IGD we branched on all the split disjunctions obtained after the reduction step described in Section 7.2 applied to all rows of the simplex tableau where the associated basic variable is a fractional integer variable. For IGD, the maximum number of rows considered in each reduction step was set to 50 (i.e.

[1] The instances are: `air01`, `air02`, `air03`, `air06`, `misc04`, `stein09`.

[2] The instances are: `atlanta-ip`, `ds`, `momentum1`, `momentum2`, `momentum3`, `msc98-ip`, `mzzv11`, `mzzv42z`, `net12`, `rd-rplusc-21`, `stp3d`.

[3] See the `miplib2003` web site: `http://miplib.zib.de/miplib2003.php`.

$M_{|R_k|} = 50 \, \forall k$). The experiments were made in a bare Branch-and-Bound setting; that is, presolving, cutting planes and heuristics were disabled. In these experiments, we chose the branching decision that closed the largest gap, regardless of the number of feasible children. In Table 7.1 we give a summary of the results for this experiment. We report the average relative integrality gap

Average closed gap (on instances with known optimum)	
Simple disjunctions (SD):	4.27%
General disjunctions (GD):	6.71%
Improved general disjunctions (IGD):	6.56%

Number of instances with largest closed gap	
Simple disjunctions (SD):	58
General disjunctions (GD):	64
Improved general disjunctions (IGD):	70

Number of instances with one child	
Simple disjunctions (SD):	10
General disjunctions (GD):	29
Improved general disjunctions (IGD):	27

Table 7.1: Results after branching at root node

closed by each method, the number of instances where each method closes at least the same absolute gap as the other two methods, and the number of instances where the disjunction that closes the largest gap generates only one feasible child. For the first criterion we only considered instances where the optimal solution is known, so that we could compute the relative amount of integrality gap close; for the remaining criteria, we also considered the instances with unknown optimum. We immediately observe that branching on general disjunctions instead of single variables yields a significantly larger amount of closed integrality gap. In our experiments, the GD method closes more gap on average than the other two methods, and both GD and IGD clearly outperform SD under this criterion. Not only GD and IGD close more gap, but they also more frequently generate only one feasible child node; the number of children was not taken into account when choosing the branching decision in this set of experiments, but it is interesting to note that with GD and IGD we often have disjunctions that close a large amount of gap and also do not increase the size of the enumeration tree. Although the GD method closes slightly more gap on average than IGD on the instances with known optimum, if we compare the number of instances where each method closes at least the same amount of

gap as the other two methods then IGD ranks first with 70 instances over 96 in total.

For many reasons, applying strong branching on all possible branching decisions is impractical, as it requires a very large computational effort which is not counterbalanced by the reduction of the size of the enumeration tree. In the remaining experiments we evaluated the performance of the branching algorithms in a framework where strong branching is applied only to the most promising branching decisions. The number of branching decisions evaluated with strong branching was set to 10. In the case of SD, we picked the 10 integer variables with the largest fractionary part (i.e. closer to 0.5). For GD and IGD, we picked the 10 split disjunctions associated with the 10 GMICs that have the largest cut off distance (equation (7.6), see (84)), where for IGD the distance was computed after the reduction step.

In the next two experiments, we solved up to 1000 nodes in the enumeration tree. We reverted back to the original branching decision selection method that favours those disjunctions which generate a smaller number of feasible children. Having a smaller number of children is a considerable advantage as we are able to progress further in depth of the enumeration tree, thus possibly leading to a larger closed gap. The evaluation criterion was the percentage of the integrality gap closed after 1000 nodes, or, if the instance was solved in less than 1000 nodes, the number of nodes required to solve to optimality. For this set of experiments, 7 instances[4] were excluded from the testbed, as solving 1000 nodes required more than 12 hours. To choose the number of rows $M_{|R_k|}$ that defines the size of the linear system at each iteration of the reduction step for the IGD method, we compared three different values: 10, 20 and 50; we included in the comparison the Reduce-and-Split (RS) coefficient improvement method introduced by Andersen, Cornuéjols and Li (8), in order to test whether their algorithm to generate good cutting planes was also suitable for branching. For fairness, we used for RS the same procedure as for the IGD methods: we picked the 10 split disjunctions associated with the 10 GMICs that have the largest cut off distance after the reduction step, and applied strong branching. We followed the implementation of the RS reduction algorithm given in the CGL library (34). A summary of the results is given in Table 7.2. It can be seen that IGD using 20 or 50 rows for the reduction step yields very similar results in terms of average closed gap on instances not solved by any method, and both choices close more gap than IGD with $M_{|R_k|} = 10$ or RS on the unsolved instances. The average number of nodes is smaller for $M_{|R_k|} = 50$. In particular IGD outperforms RS for branching. To investigate the reason for this, we recorded the average norm of the disjunction chosen for branching at each node of the enumeration tree; recall (Section 7.2.1) that the norm depends on the coefficients λ generated by the disjunction improvement algorithm. The

[4]The instances are: `dano3mip`, `fast0507`, `manna81`, `mitre`, `protfold`, `sp97ar`, `t1717`.

Number of solved instances	
RS:	36
IGD with $M_{\|R_k\|} = 10$ (IGD10):	43
IGD with $M_{\|R_k\|} = 20$ (IGD20):	42
IGD with $M_{\|R_k\|} = 50$ (IGD50):	42

Average number of nodes on instances solved by all methods	
RS:	55.7
IGD with $M_{\|R_k\|} = 10$ (IGD10):	38.4
IGD with $M_{\|R_k\|} = 20$ (IGD20):	40.4
IGD with $M_{\|R_k\|} = 50$ (IGD50):	36.5

Average gap closed on instances not solved by any method	
RS:	10.04%
IGD with $M_{\|R_k\|} = 10$ (IGD10):	13.60%
IGD with $M_{\|R_k\|} = 20$ (IGD20):	14.51%
IGD with $M_{\|R_k\|} = 50$ (IGD50):	14.32%

Number of instances with largest closed gap	
RS:	57
IGD with $M_{\|R_k\|} = 10$ (IGD10):	62
IGD with $M_{\|R_k\|} = 20$ (IGD20):	61
IGD with $M_{\|R_k\|} = 50$ (IGD50):	64

Table 7.2: Results after 1000 solved nodes

average norms (computed through a geometric mean over the whole set of instances) are as follows:

- Reduce-and-Split: 8.55;
- IGD with $M_{|R_k|} = 10$: 6.15;
- IGD with $M_{|R_k|} = 20$: 6.09;
- IGD with $M_{|R_k|} = 50$: 6.24.

Therefore, our heuristic procedure generates smaller λ's with respect to RS, which results in disjunctions that cut off a larger volume. This has a positive effect on the size of the enumeration tree. We give another possible reason for the superiority of IGD with respect to RS for branching. One of the advantages of

RS for cut generation is that the reduction algorithm generates several split disjunctions, trying to increase the distance cut off by each one of the associated cutting planes. As several cuts are generated at each round, this approach is effective (8). However, at each node of a Branch-and-Bound tree only one disjunction is chosen for branching, so a method which tries to compute only one strong disjunction is more fruitful than one that generates a set of several possibly weaker ones. This, combined with smaller coefficient λ's at the end of the reduction procedure, may explain why the algorithm described in Section 7.2 seems to be more effective than RS for branching. We decided to use IGD with $M_{|R_k|} = 50$ in all following experiments. We did not test larger values of the parameter, since solving the linear system would take too much time in practice.

A summary of the comparison between SD, GD and IGD with $M_{|R_k|} = 50$ can be found in Table 7.3. The increase in the gap per node that can be closed

Number of solved instances	
Simple disjunctions (SD):	35
General disjunctions (GD):	42
Improved general disjunctions (IGD):	42

Average number of nodes on instances solved by all methods	
Simple disjunctions (SD):	92.7
General disjunctions (GD):	52.9
Improved general disjunctions (IGD):	43.2

Average gap closed on instances not solved by any method	
Simple disjunctions (SD):	9.36%
General disjunctions (GD):	13.78%
Improved general disjunctions (IGD):	14.15%

Number of instances with largest closed gap	
Simple disjunctions (SD):	55
General disjunctions (GD):	56
Improved general disjunctions (IGD):	63

Table 7.3: Results after 1000 solved nodes

by branching on general disjunctions with respect to branching on single variables is large. This is confirmed by the results in (84). Besides, the IGD method seems to be on average superior in all respects to the other two methods, as it

closes more gap for the unsolved instances under 1000 nodes, and requires less nodes for the solved instances. This is also evident if we compare the number of instances where each method closes at least the same absolute gap as the other two methods: IGD ranks first with 63 instances over the 89 instances in the test-set. However, there are two instances where branching on simple disjunctions is more profitable than branching on general disjunctions: the `air04` and `air05` instances are solved by the SD method in 225 and 139 nodes respectively, while GD and IGD do not manage to solve them in 1000 nodes. All other instances which are solved by SD are also solved by GD and IGD.

7.4.2 Combination of several methods

Results in Table 7.3 suggest that IGD is indeed capable of closing more gap per node on a large number of instances; however, a more detailed analysis of the results shows that there are a few instances where branching on general disjunctions is not profitable, and thus both GD and IGD perform poorly. This may also happen, for example, in zero gap instances, where the enumeration of nodes with SD is usually more effective. Thus, we decided to combine both the SD and the IGD method into a single branching algorithm which tries to decide, for each instance, if it is more effective to branch on simple disjunctions or on general disjunctions. First we describe the ideas and the practical considerations behind the algorithm, and then we will describe how we implemented it.

The most evident drawback of branching on general disjunctions is that it is slower than using simple disjunctions. It is slower in several respects: the first reason is that the computations at each node take longer. This is because we have to compute the distance cut off by the GMIC associated with each row of the simplex tableau, and the reduction step that we propose involves the solution of an $M_{|R_k|} \times M_{|R_k|}$ linear system for each row which is improved, where we chose $M_{|R_k|} = 50$. All these computations are carried out several times, thus the overhead per node with respect to branching on simple disjunctions is significant. The second reason is that, by branching on general disjunction, we add one (or more) rows to the formulation of children nodes, which may result in a slowdown of the LP solution process. On the other hand, branching on simple disjunctions involves only a change in the bounds of some variables, thus the size of the LP does not increase. This suggests that branching on general disjunctions should be used only if it is truly profitable, which in turn requires a measure of profit. We decided to use the amount of closed gap as a measure of profit. Besides, since the computational overhead per node is significant when considering general disjunctions for branching, we would like to consider them only if it brings an improvement in the solution time. Thus, if on a given instance general disjunctions are never used because simple disjunctions are more profitable, we would like to disable their computation as soon as possible

Algorithm 5 CGD branching algorithm

Initialization: $ActiveGDCounter \leftarrow 3, FailedActivation \leftarrow 0, NodeCounter \leftarrow 0$
while branching **do**
  if root node **then**
    $NumGD \leftarrow 20, NumSD \leftarrow 20$
  else
    if $ActiveGDCounter > 0$ **then**
      $NumGD \leftarrow 7, NumSD \leftarrow 3$
    else
      $NumGD \leftarrow 0, NumSD \leftarrow 10$
  generate $NumGD$ general disjunctions
  generate $NumSD$ simple disjunctions
  for all branching decisions **do**
    apply strong branching
  choose a disjunction $D(\pi, \pi_0)$
  if $ActiveGDCounter > 0$ **then**
    if $D(\pi, \pi_0)$ has support > 1 **then**
      $ActiveGDCounter \leftarrow 10$
      $FailedActivation \leftarrow 0$
    else
      $ActiveGDCounter \leftarrow ActiveGDCounter - 1$
      if $ActiveGDCounter = 0$ **then**
        $FailedActivation \leftarrow FailedActivation + 1$
  else
    $NodeCounter \leftarrow NodeCounter + 1$
  if $FailedActivation < 10 \wedge NodeCounter = 100$ **then**
    $ActiveGDCounter \leftarrow 1$
    $NodeCounter \leftarrow 0$

in the enumeration tree. As the polyhedron underlying a problem may significantly change in different parts of the branching tree, it may be a good idea to test branching on general disjunctions periodically even if it has been disabled, in order to verify whether it has become profitable.

We implemented a branching algorithm based on the above considerations in the following way: at each node, branching on general disjunctions can be active or not. If it is active, we test 10 possible branching decisions with strong branching: 7 general disjunctions, and 3 simple disjunctions. General disjunctions are picked only if they generate a smaller amount of children nodes, or (in case of a tie) if the amount of closed gap is at least 50% larger. As a consequence, at all nodes where we do not manage to close any gap we always prefer simple disjunctions if they generate the same number of children as general

disjunctions. At the beginning of the enumeration tree, branching on general disjunctions is active for the first 3 nodes; moreover, we put an increased effort at root node, where we consider up to 20 simple disjunctions and 20 general disjunctions. Whenever a general disjunction is chosen for branching, then branching on general disjunctions is activated for the following 10 nodes. Otherwise, when it is disactivated (because of simple disjunctions being preferred to general disjunctions for a given number of consecutive nodes, i.e. 3 at the beginning of the enumeration tree, 10 otherwise), it is reactivated again after 100 nodes, but only for one node, in order to test whether in that part of the enumeration tree general disjunctions are worthwhile. If a general disjunction is chosen, then branching on general disjunctions is reactivated for the following 10 nodes. After 10 consecutive unfruitful activations, i.e. general disjunctions are not chosen after being activated for 10 consecutive times, branching on general disjunctions is permanently disabled. When performing the reduction step described in Section 7.2, in order to save time we do not consider all rows for reduction, but only the most promising ones. We do this by looking at the GMIC associated with each row where the basic integer variable is fractional, and sorting them by the corresponding distance cut off (7.6). The 10 rows (20 at root node) with the largest distance are modified with the reduction step of Section 7.2. Since only 7 have to be selected for strong branching, we recompute the distances and pick the 7 largest ones. We give a description of this algorithm in Algorithm 5.

To assess the practical usefulness of this approach, we compared this branching algorithm, which we will call Combined General Disjunctions (CGD), with SD. In order to evaluate the same number of branching decisions via strong branching with both methods at each node, we modified SD in order to consider, at root node, the branching decisions corresponding to the 40 integer variables with largest fractional part, and then reverting back to the usual 10 for the following nodes. We let Cplex 11.0 apply cutting planes at root node with the default parameters, and then branched for two hours. Again, preprocessing and heuristics were disabled. In Table 7.4 we compare the number of solved instances within the two hours limit, the average closed gap on instances not solved by either method, the average number of nodes and average CPU time on instances solved by both methods. The results clearly indicate that the CGD is able to combine the potential of the IGD method to close more gap with the rapidity of branching on simple disjunctions when general disjunctions are not worth the additional required time. Not only CGD solves all instances solved by SD, but it solves 3 more: `10teams` in 273.46 seconds of CPU time, `gesa2_o` in 2616.2 seconds, and `rout` in 2540.74 seconds. On the instances which have not been solved by either of the two methods, the average integrality gap closed by CGD is 31% larger in relative terms than the one closed by SD. This result is even more important if we consider that CGD is slower: in the 2 hours limit CGD solved only half as many nodes as SD on average, thus the gap closed per node

Number of solved instances	
Simple disjunctions (SD):	67
Combined general disjunctions (CGD):	70

Average number of nodes on instances solved by both methods	
Simple disjunctions (SD):	195.1
Combined general disjunctions (CGD):	98.0

Average number of nodes on instances not solved by either method	
Simple disjunctions (SD):	35796.0
Combined general disjunctions (CGD):	15075.7

Average gap closed on instances not solved by either method	
Simple disjunctions (SD):	5.35%
Combined general disjunctions (CGD):	7.03%

Average CPU time [sec] on instances solved by both methods	
Simple disjunctions (SD):	3.03
Combined general disjunctions (CGD):	3.35

Table 7.4: Results in a Branch-and-Cut framework with a two hours time limit

is significantly larger for CGD. These average values only take into account the instances with known optimum value.

We report a full table of results on the instance that have not been solved in less than two hours by the SD method in Table 7.5. If we consider the 5 instances for which the optimal solution value is not known, then on the `liu` instance both methods close the same absolute gap, on `dano3mip` CGD closes more gap, and on the remaining 3 instances (`sp97ar, t1717, timtab2`) SD closes more gap. However, on all 5 instances CGD solves a smaller amount of nodes since it is slower, and the relative difference (i.e. $\frac{\text{absolute gap SD}}{\text{absolute gap CGD}} - 1$) in closed gap on the 3 instances where SD closes more gap is small: on `timtab2`, the difference is only 0.13%, but CGD requires 4 times fewer nodes; on `sp97ar` the difference is 4.95% in favour of SD, but CGD requires 13 times fewer nodes. The difference increases on the `t1717` instance: SD closes in relative terms 12.99%

	SD ALGORITHM			CGD ALGORITHM			GAP
	CLOSED GAP			CLOSED GAP			CLOSED
INSTANCE	ABS.	REL.	NODES	ABS.	REL.	NODES	BY CUTS
10teams*	0	0%	11775	2	28.5%	398	71.3%
a1c1s1	337.58	3.21%	5340	371.423	3.54%	2578	62.29%
aflow40b	36.854	22.7%	20398	25.8243	15.9%	5477	57.3%
arki001	88.0556	6.83%	3612	580.27	45%	4000	28.27%
dano3mip	0.322586	-	8	0.374207	-	6	0%
danoint	0.310476	10.2%	5547	0.286139	9.44%	4790	2%
fast0507	0.262111	14.1%	587	0.0561795	3.03%	96	0%
gesa2_o*	84644.7	27.9%	195797	147352	48.5%	13181	51.4%
glass4	3293.85	0%	84369	3104.73	0%	79050	0%
harp2	199205	43.9%	74255	215937	47.5%	12565	32.6%
liu	214	-	108162	214	-	100347	0%
markshare1	0	0%	11027872	0	0%	2540405	0%
markshare2	0	0%	8606987	0	0%	2431791	0%
mas74	859.296	65.2%	2405902	641.509	48.7%	800207	4.6%
mkc	2.92749	6.1%	14486	6.52824	13.6%	8663	5.7%
noswot	0	0%	3192040	0	0%	1598812	0%
nsrand-ipx	158.293	6.82%	3932	222.726	9.6%	1431	49.08%
opt1217	0	0%	409010	1.33599	33.2%	316821	17%
protfold	2.32009	21.2%	140	2.14092	19.5%	150	3.6%
roll3000	127.615	7.12%	3083	293.192	16.4%	1406	40.68%
rout*	55.1337	57.6%	189312	94.9211	99.2%	28137	0.8%
set1ch	977.236	4.34%	120033	1355.82	6.02%	41034	86.06%
seymour	1.44368	7.54%	1251	1.09335	5.71%	688	41.66%
sp97ar	1.489e+06	-	4231	1.419e+06	-	318	0%
swath	28.3223	21.3%	20831	15.7973	11.9%	4724	34.9%
t1717	785.581	-	76	695.249	-	31	0%
timtab1	108754	14.8%	130014	103832	14.1%	35760	62.2%
timtab2	531157	-	50595	530454	-	12461	0%
tr12-30	183.374	0.158%	17852	691.388	0.594%	6883	99.142%

Table 7.5: Results in a Branch-and-Cut framework on the instances unsolved in two hours by the SD method. Instances with a * have been solved by the CGD method.

more gap than CGD, solving twice as many nodes in the two hours limit.

On a few instances, CGD performs strikingly better than SD. Examples are the arki001 and opt1217 instances, which are difficult instances of miplib2003. For arki001, branching with CGD closes 45% of the gap, whereas branching with SD only closes 6.83%. Similarly, for opt1217 CGD closes 33.2%, versus 0% for SD. The arki001 instance was first solved to optimality only recently by Balas and Saxena (13): they invest a large computational effort in order to generate rank-1 split cuts that close 83.05% of the integrality gap, and then use Cplex's Branch-and-Bound algorithm to close the remaining gap (16.95%) in 643425 nodes. We report that, if we run CGD on arki001 without time limits, 28.27% of the integrality gap is closed by Cplex's cutting planes with default parameters, while the remaining 71.73% is closed by our branching algorithm in 925738 nodes. Note that Balas and Saxena used the preprocessed problem as input, while in this chapter we always work with the original instances (i.e. without preprocessing). 10teams, gesa2_o, harp2, rout and tr12-30 are five other instances where CGD greatly outperforms SD. Among examples that were solved by both algorithms (see Table 7.6), bell3a required 15955 nodes using SD versus only 20 using CGD, bell5 required 773432 nodes using SD versus 24 using CGD, and gesa2 required 38539 nodes using SD versus 140 using CGD. There is also an improvement in computing time by several orders of magnitude on these three instances.

On those instances which are solved by both methods, CGD requires on average only half the nodes needed by SD, and the average CPU time is very close for both methods (with a slight advantage for SD). Full results are reported in Table 7.6.

Summarizing, in our experiments the combination between SD and IGD, which we have called CGD, seems clearly superior to the traditional branching strategy that is represented by branching on single variables. Moreover, as Cplex's callable library is not optimized for branching on general disjunctions, the implementation of CGD could be made faster.

	Gap Closed			SD algorithm		CGD algorithm	
	By Cuts	By Branching			Time		Time
Instance		Abs.	Rel.	Nodes	[sec]	Nodes	[sec]
aflow30a	65.9%	59.6358	34.1%	1813	77.886	1725	99.839
air04	17.9%	494.084	82.1%	181	164.972	203	683.874
air05	15.1%	421.787	84.9%	209	105.902	241	133.679
bell3a	70.8%	4638.26	29.2%	15955	11.822	20	0.047
bell3b	89.6%	39855.3	10.4%	1206	2.177	526	5.512
bell4	91.93%	44957.8	8.07%	9091	24.177	3636	24.242
bell5	85.6%	51456.8	14.4%	773432	553.703	24	0.128
blend2	23.2%	0.524858	76.8%	539	5.321	454	9.920
bm23	24.8%	10.0974	75.2%	119	0.272	78	0.364
cap6000	37.6%	113.47	62.4%	289	30.176	236	111.553
dcmulti	68.5%	1323.83	31.5%	41	1.050	56	2.853

	Gap Closed			SD algorithm		CGD algorithm	
	By Cuts	By Branching			Time		Time
Instance		Abs.	Rel.	Nodes	[sec]	Nodes	[sec]
dsbmip	100%	0	0%	15	1.666	23	2.754
egout	35.7%	568.101	64.3%	1	0.009	1	0.011
fiber	91.83%	20400.8	8.17%	153	3.944	28	4.025
fixnet3	97.98%	227.43	2.02%	5	0.300	5	0.421
fixnet4	87.7%	573.738	12.3%	33	1.438	52	8.526
fixnet6	83.4%	461.791	16.6%	1087	20.417	1365	52.859
flugpl	11.8%	30286.3	88.2%	199	0.074	16	0.021
gen	100%	112313	0%	0	0.021	0	0.026
gesa2	74.9%	76271.3	25.1%	38539	1232.150	140	28.490
gesa3	69.3%	48425.7	30.7%	51	2.149	63	4.016
gesa3_o	70.9%	45960.5	29.1%	89	3.934	34	9.955
gt2	91.65%	643.634	8.35%	236	0.412	43	0.139
khb05250	99.9336%	7317.49	0.0664%	5	0.106	2	0.105
l152lav	30.1%	65.4949	69.9%	552	15.614	149	16.251
lp4l	76%	5.875	24%	3	0.059	3	0.160
lseu	68.1%	91.0289	31.9%	61	0.181	46	0.349
manna81	100%	0	0%	0	0.143	0	0.147
mas76	4.2%	1065.02	95.8%	309659	651.702	377398	2608.890
misc01	44.5%	281.057	55.5%	251	3.836	274	7.151
misc02	56.6%	295.312	43.4%	19	0.148	10	0.191
misc03	9.8%	1308.17	90.2%	255	3.73	496	11.875
misc05	45.2%	29.3913	54.8%	103	1.658	33	0.823
misc06	26.5%	6.83269	73.5%	17	1.110	17	2.365
misc07	5.8%	1313.75	94.2%	12139	462.649	25940	1536.48
mitre	100%	0	0%	15	4.394	15	10.759
mod008	21.9%	12.5493	78.1%	345	0.937	13	0.095
mod010	28%	11.5	72%	25	0.567	2	0.440
mod011	68.2%	2.40503e+06	31.8%	707	2633.340	250	3539.000
mod013	30.1%	17.4348	69.9%	115	0.317	107	0.422
modglob	73.7%	81583	26.3%	1879	48.692	2387	75.699
nw04	9.1%	501.358	90.9%	83	75.879	48	109.610
p0033	99.9159%	0.478261	0.0841%	3	0.005	3	0.007
p0040	100%	62027	0%	0	0.002	0	0.001
p0201	46%	400	54%	69	1.147	50	1.635
p0282	96.99%	2458.44	3.01%	23	0.218	12	0.261
p0291	48.5%	5223.75	51.5%	0	0.017	0	0.018
p0548	99.9274%	6.08471	0.0726%	9	0.076	6	0.157
p2756	98.49%	6.56956	1.51%	7	0.364	13	1.205
pipex	63.5%	5.30334	36.5%	19	0.041	12	0.050
pk1	0%	11	100%	243317	956.355	189740	1468.170
pp08aCUTS	87.1%	240.666	12.9%	711	12.363	658	18.583
pp08a	94.38%	258.537	5.62%	392	4.633	372	4.481
qiu	0%	798.766	100%	19399	2780.000	19399	2901.890
qnet1	71%	509.709	29%	53	3.156	74	26.939
qnet1_o	85.1%	585.272	14.9%	17	1.267	13	3.826
rentacar	51%	759381	49%	11	12.047	11	14.973
rgn	15.9%	28.0903	84.1%	2089	2.143	1703	3.826
sample2	46.5%	68.4556	53.5%	35	0.092	33	0.103
sentoy	24.9%	50.6089	75.1%	52	0.175	53	0.266

	Gap Closed			SD algorithm		CGD algorithm	
	By Cuts	By Branching			Time		Time
Instance		Abs.	Rel.	Nodes	[sec]	Nodes	[sec]
set1al	99.9521%	2.2619	0.0479%	5	0.056	6	0.145
set1cl	34.7%	6484.25	65.3%	0	0.021	0	0.023
stein15	0%	2	100%	42	0.058	44	0.068
stein27	0%	5	100%	1628	3.785	1537	3.721
stein45	0%	8	100%	29676	218.862	28882	215.015
vpm1	89.1%	0.5	10.9%	17	0.092	17	0.107
vpm2	77%	0.888645	23%	1299	15.646	477	5.723

Table 7.6: Results in a Branch-and-Cut framework on the instances solved by both the SD and the CGD method.

7.5 Computational experiments: MILP formulation

We implemented a branching scheme based on the formulation described in Section 7.3 with the neighbourhood search algorithm of Section 7.3.1 within Cplex 11.0 (83). In this section we provide preliminary computational experiments.

We modified Cplex's branching algorithm so to execute, at each node of the numeration tree, the following steps:

1. Set up the auxiliary problem (7.29) using the optimal basis of the LP relaxation at the current node
2. Solve the auxiliary problem (7.29) with Cplex's branch-and-cut algorithm for a limited number of nodes, emphasizing the use of heuristics, to obtain an initial set of solutions
3. Apply Algorithm 4 to generate more solutions via local search
4. Apply strong branching to all candidates to select the best disjunction with respect to a given criterion

We set $K = 10$ and $\eta = 0.05$ in (7.29). We recall that K is an upper bound on the 1-norm of the disjunctions, and η is the minimum amount by which each disjunction should be violated. At step 2), we limit the solution process to 60 seconds or 1000 nodes, whichever comes first. The maximum number of disjunctions generated by Algorithm 4 was set to $M = 20$. Let x^1, x^2 be the optimal solutions to the LP relaxations of the children that would be generated by branching on a disjunction; we select the disjunction that maximizes $\min(c^\top \bar{x}^1, c^\top \bar{x}^2)$ among all candidates.

In the formulation of the auxiliary problem (7.29), some changes can help decreasing the computational burden. First, since $\forall i \in N \setminus N_I$ we must have $\pi_i = 0$, we can drop the variables π_i for $i \in N \setminus N_I$. Next, the variables $\pi_i, i \in N_I$ can be dropped and replaced by $\pi_i^+ - \pi_i^-$. Finally, the variables $w_j \forall j \in J$ can be dropped, and the constraints $\forall j \in J\ w_j \leq zc^\top r^j, \forall j \in J\ w_j \geq -\pi^\top r^j, \forall j \in J\ w_j \geq \pi^\top r^j$ can be replaced with $\forall j \in J\ zc^\top r^j \geq -\pi^\top r^j, \forall j \in J\ zc^\top r^j \geq \pi^\top r^j$.

In the following, we apply different branching methods to a set of 78 heterogeneous instances taken from `miplib2.0`, `miplib3`, `miplib2003`. Cutting planes and heuristics are disabled; thus, we apply a pure branch-and-bound scheme for 1000 nodes, and compare the amount of integrality gap closed after 1000 nodes or, in case the instance is solved before this limit, the number of nodes in the enumeration tree. The split disjunctions are generated with three different methods: the one described above based on the formulation of Section 7.3, which we call the MILP method; the heuristic procedure with a quadratic formulation described in Section 7.4.2, which we call the IGD method; the traditional branching scheme of selecting integer variables with fractional values in the current solution, and considering the corresponding simple disjunction, which we call the SD method.

Table 7.7 summarizes the results; details can be found in Table 7.8. All averages are geometric. For the SD method, strong branching was applied to *all* integer variables with fractional value. As confirmed by Section 7.4, the average values clearly indicate that branching on general disjunctions is very effective if compared to branching on simple disjunctions. The number of nodes for the instances solved by all methods is significantly smaller for the IGD method, which in general seems to perform better than SD and MILP. In particular, IGD solves 3 more instances with respect to MILP; these are the `bell` instances, where MILP does not perform as well as IGD. Moreover, IGD requires significantly fewer nodes than SD and MILP on the instances which are solved by all methods. On the other hand, MILP ranks first in terms of gap closed on the unsolved instances, which suggests that MILP shows good results on the difficult instances. MILP also ranks first if comparing the number of instances on which each method closes an absolute gap which is at least as large as the gap closed by the other two methods. Indeed, MILP closes more gap than IGD and SD on the following instances with medium to hard difficulty: `a1c1s1`, `arki001`, `fixnet3`, `fixnet6`, `gesa2`, `gesa2_o`, `mkc`, `pp08aCUTS`, `pp08a`, `tr12-30`. However, MILP requires significantly more time: on average, on the instances which are unsolved in less than 1000 nodes by neither MILP nor IGD, MILP requires 4700 seconds of CPU time, whereas IGD requires only 45. We note that, although the neighbourhood search phase should ideally iterate until $M = 20$ split disjunctions are generated, the arithmetic average number of disjunctions generated each node is 16.36. This indicates that the neighbourhood search algorithm often terminates because of lack of starting solutions.

Number of solved instances	
MILP formulation:	35
IGD with $M_{\lvert R_k \rvert} = 50$:	38
SD full strong branching:	30

Average number of nodes on instances solved by all methods	
MILP formulation:	86.19
IGD with $M_{\lvert R_k \rvert} = 50$:	44.97
SD full strong branching:	105.07

Average gap closed on instances not solved by any method	
MILP formulation:	13.37%
IGD with $M_{\lvert R_k \rvert} = 50$:	11.10%
SD full strong branching:	11.71%

Number of instances with largest closed gap	
MILP formulation:	52
IGD with $M_{\lvert R_k \rvert} = 50$:	50
SD full strong branching:	43

Table 7.7: Results after 1000 solved nodes

	MILP formulation (MILP)				Simple disjunctions (SD)				Quadratic formulation (IGD)			
	Gap	% Gap	Nodes	Time	Gap	% Gap	Nodes	Time	Gap	% Gap	Nodes	Time
a1c1s1	1177.15	11.2%	1000	8294.63	598.96	5.7%	1000	884.044	174.912	1.66%	1000	6806.43
aflow30a	59.8128	34.2%	1000	13381.3	75.0668	42.9%	1000	83.7643	76.8047	43.9%	1000	158.725
arki001	681.547	58.7%	1000	20938.6	521.32	46.3%	1000	4395.55	102.853	13.9%	1000	1718.05
bell3a	7330.31	90%	1000	609.854	5238.26	76.8%	1000	0.754885	8915.19	100%	33	0.063991
bell3b	45866.2	94.9%	1000	733.798	61748.2	99%	1000	4.47032	65378.7	100%	84	0.895863
bell4	28850.5	93.8%	1000	677.617	41535.9	96%	1000	7.54185	28297.7	93.7%	1000	11.9222
bell5	59979.6	100%	185	61.4357	50302.4	97.3%	1000	1.61875	59979.6	100%	33	0.229965
blend2	0.677565	100%	274	630.595	0.677565	100%	354	5.57815	0.677565	100%	210	13.7489
bm23	13.4291	100%	92	43.5834	13.4291	100%	85	0.076988	13.4291	100%	43	0.106984
cap6000	166.325	100%	305	45681.4	176.325	100%	434	48.8776	178.325	100%	356	171.701
danoint	0.111858	3.69%	1000	25424.8	0.117069	3.86%	1000	3025.65	0.100749	3.32%	1000	1854.24
dcmulti	2589.02	100%	599	550.045	2589.02	100%	643	3.65744	437.92	48.8%	1000	54.6597
dsbmip	0	100%	31	306.658	0	100%	23	53.8688	0	100%	26	34.1718
egout	211.412	100%	894	589.382	157.219	87%	1000	1.06584	174.833	91.3%	1000	1.78673
fiber	142909	57.7%	1000	23727.3	106741	43.2%	1000	46.8629	248717	100%	463	57.0363
fixnet3	3338.5	30.2%	1000	1922.01	2674.62	24.3%	1000	47.9247	2075.4	19%	1000	116.819
fixnet4	944.888	20.2%	1000	3258.89	956.65	20.4%	1000	60.4988	426.406	9.11%	1000	125.82
fixnet6	719.252	25.9%	1000	3840.7	346.772	12.5%	1000	66.9868	241.936	8.7%	1000	122.595
flugpl	33624.8	100%	137	28.6606	33624.8	100%	931	0.331949	33624.8	100%	21	0.019997
gen	84.7312	100%	26	27.8918	84.7312	100%	138	2.37364	84.7312	100%	11	0.742887
gesa2	260466	85.8%	1000	2562.12	164794	54.3%	1000	96.8013	52017.3	17.1%	1000	281.627
gesa2_o	217878	71.8%	1000	4494.92	164794	54.3%	1000	93.6328	48627.8	16%	1000	288.23
gesa3	157410	100%	572	1481.36	157410	100%	258	31.5362	157410	100%	591	122.873

	MILP formulation (MILP)				Simple disjunctions (SD)				Quadratic formulation (IGD)			
	Gap	% Gap	Nodes	Time	Gap	% Gap	Nodes	Time	Gap	% Gap	Nodes	Time
gesa3_o	157410	100%	432	2031.91	157410	100%	286	29.0646	157410	100%	958	181.184
glass4	7.987e-06	0%	1000	6069.47	1100	0%	1000	41.3407	7.987e-06	0%	1000	35.0307
gt2	7705.77	100%	193	425.666	7705.77	100%	99	0.164974	7705.77	100%	34	0.153976
harp2	180686	46%	1000	102603	203908	51.1%	1000	36.4025	123485	33.4%	1000	121.588
khb05250	1.102e+07	100%	921	1022.84	1.102e+07	100%	723	10.6674	1.102e+07	100%	611	17.7793
liu	214	-	1000	21114.4	214	-	1000	284.748	0	-	1000	723.5
lp4l	5.18831	100%	4	94.5356	5.18831	100%	4	0.098984	5.18831	100%	2	0.086986
lseu	285.318	100%	828	922.694	171.462	60.1%	1000	1.11083	285.318	100%	64	0.364945
manna81	1	0.751%	1000	85790.8	1.5	1.13%	1000	6193.58	0.5	0.376%	298	14436.2
markshare1	0	0%	1000	2070.21	0	0%	1000	0.670898	0	0%	1000	8.66268
markshare2	0	0%	1000	3715.36	0	0%	1000	0.894863	0	0%	1000	11.9542
mas74	172.912	13.1%	1000	6548.5	200.418	15.2%	1000	6.32704	221.672	16.8%	1000	13.9729
mas76	193.882	17.4%	1000	5344.13	180.767	16.3%	1000	4.42533	244.732	22%	1000	10.8833
misc01	506.5	100%	111	171.165	506.5	100%	137	0.85387	506.5	100%	83	1.3138
misc02	680	100%	16	7.2159	680	100%	15	0.045993	680	100%	11	0.096985
misc03	1450	100%	159	558.287	1450	100%	138	2.97655	1450	100%	188	7.15491
misc05	53.6	100%	22	18.6892	53.6	100%	72	0.942856	53.6	100%	44	1.50777
misc06	9.17135	100%	43	58.8371	9.17135	100%	20	0.492925	9.17135	100%	16	1.36079
misc07	215.714	15.5%	1000	16089.5	51.6667	3.7%	1000	203.662	119.167	8.54%	1000	155.163
mkc	39.38	82%	1000	87308.4	38.43	80%	1000	404.27	30	62.5%	1000	3764.3
mod008	16.0689	100%	361	1877.13	9.865	61.4%	1000	1.16782	16.0689	100%	17	0.061991
mod010	15.9167	100%	9	1122.47	15.9167	100%	23	1.08083	15.9167	100%	3	0.45993
mod013	24.9333	100%	185	149.338	24.9333	100%	227	0.177972	24.9333	100%	54	0.141979
modglob	131127	42.3%	1000	1608.1	131595	42.5%	1000	51.8241	85446.9	27.6%	1000	41.4137
noswot	0	0%	1000	944.83	0	0%	1000	3.17852	0	0%	1000	12.2511
opt1217	0.272819	6.78%	1000	80674.2	0	0%	1000	13.47	1.92367	47.8%	1000	51.3492
p0033	568.428	100%	143	69.2525	568.428	100%	261	0.06399	568.428	100%	21	0.016997
p0040	225.618	100%	6	2.76158	225.618	100%	31	0.009999	225.618	100%	2	0.002
p0201	740	100%	106	602.537	740	100%	70	1.40979	740	100%	40	1.15882
p0282	81543.5	100%	72	77.1203	81543.5	100%	42	0.162975	81543.5	100%	57	0.478928
p0291	2994.65	100%	18	13.9899	2994.65	100%	21	0.050992	2994.65	100%	12	0.047993
p0548	8136.34	100%	762	2550.91	8136.34	100%	784	4.17037	8136.34	100%	121	2.25266
p2756	3.21948	3.08%	1000	31487.3	2.19718	2.85%	1000	87.2827	3.52113	3.15%	1000	292.204
pipex	14.5119	100%	71	62.9474	14.5119	100%	325	0.167974	14.5119	100%	13	0.029996
pk1	0	0%	1000	2695.28	0	0%	1000	7.61684	0	0%	1000	23.2875
pp08aCUTS	680.439	36.4%	1000	2422	612.248	32.7%	1000	61.4507	417.95	22.4%	1000	45.8
pp08a	1332.14	28.9%	1000	901.196	1236	26.9%	1000	13.7629	1128.5	24.5%	1000	23.1855
rgn	33.4	100%	847	816.246	33.4	100%	859	0.898863	33.4	100%	568	1.44878
rout	14.7457	15.4%	1000	27213.4	19.8707	20.8%	1000	92.168	53.4079	55.8%	1000	116.207
sample2	128	100%	157	46.4479	128	100%	93	0.054991	128	100%	87	0.105984
sentoy	67.278	100%	63	60.4588	67.278	100%	54	0.072989	67.278	100%	35	0.122982
set1al	364.408	17.3%	1000	1513.83	423.136	18.6%	1000	147.961	312.342	16.2%	1000	260.723
set1ch	2102.43	23.1%	1000	2664.5	1842.41	22%	1000	147.174	1244.09	19.3%	1000	174.753
set1cl	359.619	18.1%	1000	1311.85	356.628	18%	1000	113.272	300.171	16.9%	1000	259.563
stein15	2	100%	38	7.61984	2	100%	44	0.069989	2	100%	31	0.066989
stein27	2	40%	1000	294.986	3	60%	1000	4.03639	5	100%	865	3.42748
stein45	1	12.5%	1000	788.01	2.33333	29.2%	1000	39.0691	2.09804	26.2%	1000	23.8184
timtab1	53915	7.32%	1000	3395.07	119678	16.3%	1000	29.0176	123309	16.8%	1000	58.1322
timtab2	14394	-	1000	7475.45	83794	-	1000	92.131	93552	-	1000	211.767
tr12-30	3666.48	6.51%	1000	3079.88	3014.82	5.95%	1000	1169.3	1289.27	4.47%	1000	958.276
vpm1	4.58333	100%	21	26.9529	1.15	25.1%	1000	6.46402	4.58333	100%	25	0.259961
vpm2	1.39564	39.6%	1000	2057.29	1.34673	38.3%	1000	14.6928	1.50242	42.4%	1000	27.2479

Table 7.8: Results after 1000 solved nodes: full results

Several reasons suggest that MILP may be better used to generate cutting planes, instead of branching decisions. The neighbourhood search algorithm generates a pool of solutions, but then lacks new starting solutions to continue

the search. This problem could be avoided by adding to the original problem the intersection cuts associated to the generated split disjunctions, and resolving the LP. Moreover, the observation of the solution process with Cplex suggest that the MILP formulation works better at the beginning of the branching tree, because the auxiliary problem (7.29) which we have to solve at each node grows in size as we progress in the enumeration. The analysis of the strength of the cutting planes as we spend more time in finding good solutions to the auxliary problem (7.29), and the effect of changing the parameter η, are interesting subjects for future research.

Chapter 8

A Good Recipe for Solving MINLPs

In Section 1.3.2 we proposed a model of the time-dependent shortest path problem on possibly non FIFO networks with arbitrary arc cost functions, which turns out to be a MINLP. If we model the effect of traffic congestions on travelling times as a summation of Gaussian functions, then that formulation is nonconvex (see Section 1.3.3. From the modelling point of view, nonconvex MINLPs are the most expressive mathematical programs. It stands to reason that general-purpose MINLP solvers should be very useful. Currently, optimal solutions of MINLPs in general form are obtained by using the spatial Branch-and-Bound (sBB) algorithm (3; 92; 131; 132); but guaranteed optima can only be obtained for relatively small-sized MINLPs. Realistically-sized MINLPs can often have thousands (or tens of thousands) of variables (continuous and integer) and nonconvex constraints. With such sizes, it becomes a hard challenge to even find a feasible solution, and sBB algorithms become almost useless. Some good solvers targeting convex MINLPs exist in the literature (2; 24; 26; 57; 58; 90); although they can all be used on nonconvex MINLPs as well (forsaking the optimality guarantee), in practice their mileage varies wildly with the instance of the problem being solved, resulting in a high fraction of "false negatives" (i.e. feasible problems for which no feasible solution was found). The Feasibility Pump (FP) idea was recently extended to convex MINLPs (25), but again this does not work so well when applied to nonconvex MINLPs unmodified (101).

In this section, we propose an effective and reliable MINLP heuristic, called the Relaxed-Exact Continuous-Integer Problem Exploration (RECIPE) algorithm. The MINLPs we address are cast in the following general form:

$$\left.\begin{array}{lrcl} \min\limits_{x\in\mathbb{R}^n} & & f(x) & \\ \text{s.t.} & l \leq & g(x) & \leq u \\ & x^L \leq & x & \leq x^U \\ & x_i \in \mathbb{Z} & & \forall\, i \in Z \end{array}\right\} \tag{8.1}$$

In the above formulation, x are the decision variables (x_i is integer for each

$i \in Z$ and continuous for each $i \notin Z$, where $Z \subseteq \{1, \ldots, n\}$). $f : \mathbb{R}^n \to \mathbb{R}$ is a possibly nonlinear function, $g : \mathbb{R}^n \to \mathbb{R}^m$ is a vector of m possibly nonlinear functions (assumed to be differentiable), $l, u \in \mathbb{R}^m$ are the constraint bounds (which may be set to $\pm\infty$), and $x^L, x^U \in \mathbb{R}^n$ are the variable bounds.

RECIPE puts together a global search phase based on Variable Neighbourhood Search (VNS) (76) and a local search phase based on a Branch-and-Bound (BB) type heuristic. The VNS global phase rests on neighbourhoods defined as hyperrectangles for the continuous and general integer variables (95) and by Local Branching (LB) for the binary variables (56). The local phase employs a BB solver for convex MINLPs (58), and applies it to (possibly nonconvex) MINLPs, making therefore effectively a heuristic. A local NLP solver, which implements a Sequential Quadratic Programming (SQP) algorithm (66), supplies an initial constraint-feasible solution to be employed by the BB as starting point. RECIPE is an efficient, effective and reliable general-purpose algorithm for solving complex MINLPs of small and medium scale.

The rest of this section is organized as follows. In Section 8.1 we describe the basic component algorithms on which RECIPE is based. Section 8.2 presents the overall approach. In Section 8.3 we discuss computational results obtained over MINLPLib, focusing on optimality, reliability and speed.

8.1 The basic ingredients

This section describes the four main components used in RECIPE, which are:

- the global search phase: Variable Neighbourhood Search;
- the binary variable neighbourhood definition technique: Local Branching;
- the constraint and integral feasibility enforcing local solution algorithm: Branch-and-Bound for cMINLPs;
- the constraint feasibility enforcing local solution algorithm: Sequential Quadratic Programming.

8.1.1 Variable neighbourhood search

VNS relies on iteratively exploring neighbourhoods of growing size to identify better local optima (76; 77; 78). More precisely, VNS escapes from the current local minimum x^* by initiating other local searches from starting points sampled from a neighbourhood of x^* which increases its size iteratively until a local minimum better than the current one is found. These steps are repeated until a given termination condition is met. This can be based on CPU time, number of non-improving steps and other configurable parameters.

VNS has been applied to a wide variety of problems both from combinatorial and continuous optimization (9; 27; 52; 89; 96; 97; 122). Its early applications to continuous problems were based on a particular problem structure. In the continuous location-allocation problem, the neighbourhoods are defined according to the meaning of problem variables (assignments of facilities to customers, positioning of yet unassigned facilities and so on) (27). In bilinearly constrained bilinear problems the neighbourhoods are defined in terms of the applicability of the successive linear programming approach, where the problem variables can be partitioned so that fixing the variables in either set yields a linear problem; more precisely, the neighbourhoods of size k are defined as the vertices of the LP polyhedra that are k pivots away from the current vertex (76). The first VNS algorithm targeted at problems with fewer structural requirements, namely, box-constrained nonconvex NLPs, was given in (105) (the paper focuses on a particular class of box-constrained NLPs, but the proposed approach is general). Its implementation is described in (51). Since the problem is assumed to be box-constrained, the neighbourhoods arise naturally as hyperrectangles of growing size centered at the current local minimum x^*. The same neighbourhoods were used in (95), an extension of VNS to constrained NLPs.

8.1.2 Local branching

LB is an efficient heuristic for solving difficult Mixed-Integer Linear Programming (MILP) problems (56); see also Section 7.3.1. Given an integer $k > 0$, the LB search explores k-neighbourhoods of the incumbent x^* by allowing at most k of the integer variables to change their value; this condition is enforced by means of the *local branching constraint*:

$$\sum_{i \in \bar{S}} (1 - x_i) + \sum_{i \notin \bar{S}} x_i \leq k, \tag{8.2}$$

where $\bar{S} = \{i \leq n \mid i \in Z \wedge x_i^* = 1\}$, which defines a neighbourhood of radius k with respect to the binary variables of (8.1), centered at a binary solution with support $\bar{S}$. LB updates the incumbent as it finds better solutions. When this happens, the LB procedure is called iteratively with $\bar{S}$ relative to the new incumbent. We remark that LB was successfully used in conjunction with VNS in (79).

8.1.3 Branch-and-bound for cMINLPs

Solving cMINLPs (i.e. MINLPs where the objective function and constraints are convex — the terminology is confusing as all MINLPs are actually nonconvex problems because of the integrality constraints) is conceptually not much more

difficult than solving MILPs: as the relaxed problem is convex, obtaining lower bounds is easy. The existing tools, however, are still far from the quality attained by modern MILP solvers. The problem is usually solved by BB, where only the integer variables are selected for branching. A restricted (continuous) convex NLP is formed and solved at each node, where the variable ranges have been restricted according to the node's definition. Depending on the algorithm, the lower bounding problem at each node may either be the original problem with relaxed integrality constraints (35; 58) (in which case the BB becomes a recursive search for a solution that is both integer feasible and a local optimum in continuous space), or its linear relaxation by outer approximation (2; 24; 54; 57). In the former case, the restricted NLP is solved to optimality at each node by using local NLP methods (which converge to the node's global optimum when the problem is convex) such as SQP (see Sect. 8.1.4), in the latter it is solved once in a while to get good incumbent candidates.

Another approach to solving MINLPs, which can be applied to convex and pseudoconvex objective and constraints alike, is taken in (135; 136; 137), where a cutting planes approach is blended in with a sequence of MILP subproblems (which only need to be solved to feasibility).

These approaches guarantee an optimal solution if the objective and constraints are convex, but may be used as a heuristic even in presence of nonconvexity. Within this section, we employ these methods in order to find local optima of general (nonconvex) MINLPs. The problem of finding an initial feasible starting point (used by the BB local NLP subsolver) is addressed by supplying the method with a constraint feasible (although not integer feasible) starting point found by an SQP algorithm (see Sect. 8.1.4).

8.1.4 Sequential quadratic programming

SQP methods find local solutions to nonconvex NLPs. They solve a sequence of quadratic approximations of the original problem subject to a linearization of its constraints. The quadratic approximation is obtained by a convex model of the objective function Hessian at a current solution point, subject to a linearization of the (nonlinear) constraints around the current point. SQP methods are now at a very advanced stage (66), with corresponding implementations being able to warm- or cold-start. In particular, they deal with the problem of infeasible linear constraints (this may happen as the linearization around a point of a set of feasible nonlinear constraints is not always feasible), as well as the feasibility of the starting point with respect to the nonlinear constraints. This case is dealt with by elastic programming (65). In particular, SNOPT does a good job of finding a constraint feasible point out of any given initial point, even for reasonably large-scale NLPs. By starting a local MINLP solver from a constraint feasible starting point, there are better chances that an integer feasible solution may be found.

8.2 The RECIPE algorithm

Our main algorithm is a heuristic exploration of the problem solution space by means of an alternating search between the relaxed NLP and the exact MINLP. This is a two-phase global optimization method. Its local phase consists in using the SQP algorithm for solving relaxed (nonconvex) NLPs locally; next, the BB algorithm is used for solving exact (nonconvex) MINLPs to feasibility. The global phase of the algorithm is given by a Variable Neighbourhood Search using two separate neighbourhoods for continuous and general integer variables and for binary variables. The former neighbourhoods have hyper-rectangular shape; the latter are based on a LB constraint involving all binary variables.

We consider a (nonconvex) MINLP P given by formulation (8.1), with its continuous relaxation $\bar{P}$. Let $B = \{i \in Z \mid x_i^L = 0 \wedge x_i^U = 1\}$ be the set of indices of the binary variables, and $\bar{B} = \{1, \ldots, n\} \setminus B$ the set of indices of others, including general integer and continuous variables. Let $Q(\bar{x}, k, k_{\max})$ be its reformulation obtained by adding a local branching constraint

$$\sum_{i \in B} (\bar{x}_i(1 - x_i) + (1 - \bar{x}_i)x_i) \leq \left\lceil k \frac{|B|}{k_{\max}} \right\rceil, \tag{8.3}$$

where $\bar{x}$ is a (binary) feasible solution (e.g. obtained at a previous iteration), $k_{\max} \in \mathbb{N}$ and $k \in \{1, \ldots, k_{\max}\}$. At each VNS iteration (with a certain associated parameter k), we obtain an initial point $\tilde{x}$, where $\tilde{x}_i$ is sampled in a hyperrectangular neighbourhood of radius k for $i \in \bar{B}$ (rounding where necessary for $i \in Z \setminus B$) and $\tilde{x}_i$ is chosen randomly for $i \in B$. We then solve the continuous relaxation $\bar{P}$ locally by means of an SQP method using $\tilde{x}$ as a starting point, and obtain $\bar{x}$ (if $\bar{x}$ is not feasible with respect to the constraints of P, then $\bar{x}$ is re-set to $\tilde{x}$ for want of a better choice). We then use a BB method for cMINLPs in order to solve $Q(\bar{x}, k, k_{\max})$, obtaining a solution x'. If x' improves on the incumbent x^*, then x^* is replaced by x' and k is reset to 1. Otherwise (i.e. if x' is worse than x^* or if $Q(\bar{x}, k, k_{\max})$ could not be solved), k is increased in a VNS-like fashion. The algorithm is described formally in Alg. 6.

8.2.1 Hyperrectangular neighbourhood structure

We discuss here the neighbourhood structure for $N_k(x)$ for the RECIPE algorithm.

Consider hyperrectangles $H_k(x)$, centered at $x \in \mathbb{R}^n$ and proportional to the hyperrectangle $x^L \leq x \leq x^U$ given by the original variable bounds, such that $H_{k-1}(x) \subset H_k(x)$ for each $k \leq k_{\max}$. More formally, let $H_k(x^*)$ be the hyperrect-

Algorithm 6 The RECIPE algorithm.

INPUT: Neighbourhoods $N_k(x)$ for $x \in \mathbb{R}^n$;
maximum neighbourhood radius $k_{\max}$;
number L of local searches in each neighbourhood.
OUTPUT: Best solution found x^*.
Set $x^* = x^L/2 + x^U/2$
while (!time-based termination condition) **do**
  Set $k \leftarrow 1$
  while ($k \leq k_{\max}$) **do**
    for ($i = 1$ to L) **do**
      Sample a random point $\tilde{x}$ from $N_k(x^*)$.
      Solve $\bar{P}$ using an SQP algorithm from initial point $\tilde{x}$ obtaining $\bar{x}$
      if ($\bar{x}$ is not feasible w.r.t. the constraints of P) **then**
        $\bar{x} = \tilde{x}$
      Solve $Q(\bar{x}, k, k_{\max})$ using a BB algorithm from initial point $\bar{x}$ obtaining x'
      if (x' is better than x^*) **then**
        Set $x^* \leftarrow x'$
        Set $k \leftarrow 0$
        Exit the FOR loop
    Set $k \leftarrow k + 1$.

angle $y^L \leq x \leq y^U$ where, for all $i \notin Z$,

$$\begin{aligned} y_i^L &= x_i^* - \frac{k}{k_{\max}}(x_i^* - x_i^L) \\ y_i^U &= x_i^* + \frac{k}{k_{\max}}(x_i^U - x_i^*), \end{aligned}$$

for all $i \in Z \setminus B$,

$$\begin{aligned} y_i^L &= \lfloor x_i^* - \frac{k}{k_{\max}}(x_i^* - x_i^L) + 0.5 \rfloor \\ y_i^U &= \lfloor x_i^* + \frac{k}{k_{\max}}(x_i^U - x_i^*) + 0.5 \rfloor, \end{aligned}$$

and for all $i \in B$, $y^L = 0$ and $y^U = 1$.

We let $N_k(x) = H_k(x) \backslash H_{k-1}(x)$. This neighbourhood structure defines a set of hyperrectangular nested shells with respect to continuous and general integer variables. Let τ be the affine map sending the hyperrectangle $x^L \leq x \leq x^U$ into the unit L_∞ ball (i.e., hypercube) $\mathcal{B}$ centered at 0, i.e., $\mathcal{B} = \{x : |x_i| \leq 1 \forall i\}$. Let $r_k = \frac{k}{k_{\max}}$ be the radii of the balls $\mathcal{B}_k$ (centered at 0) such that $\tau(H_k(x)) = \mathcal{B}_k$ for each $k \leq k_{\max}$. In order to sample a random vector $\tilde{x}$ in $\mathcal{B}_k \backslash \mathcal{B}_{k-1}$ we proceed as in Alg. 7.

Algorithm 7 Sampling in the shell neighbourhoods.

INPUT: $k, k_{\max}$.
OUTPUT: A point $\tilde{x}$ sampled in $H_k(x) \setminus H_{k-1}(x)$.
Sample a random direction vector $d \in \mathbb{R}^n$
Normalize d (i.e., set $d \leftarrow \frac{d}{||d||_\infty}$)
Let $r_{k-1} = \frac{k-1}{k_{\max}}$, $r_k = \frac{k}{k_{\max}}$
Sample a random radius $r \in [r_{k-1}, r_k]$ yielding a uniformly distributed point in the shell
Let $\tilde{x} = \tau^{-1}(rd)$

The sampled point $\tilde{x}$ will naturally not be feasible in the constraints of (8.1), but we can enforce integral feasibility by rounding $\tilde{x}_j$ to the nearest integer for $j \in Z$, i.e. by setting $\tilde{x}_j \leftarrow \lfloor \tilde{x}_j + 0.5 \rfloor$. This will be rather ineffective with the binary variables x_j, which would keep the same value $\tilde{x}_j = x^*_j$ for each $k \leq \frac{k_{\max}}{2}$. Binary variables are best dealt with by solving the LB reformulation Q in Alg. 6.

8.3 Computational results

Alg. 6 presents many implementation difficulties: the problem must be reformulated iteratively with the addition of a different LB constraint at each iteration; different solvers acting on different problem formulations must be used. All this must be coordinated by the outermost VNS at the global level. We chose AMPL (61) as a scripting language because it makes it very easy to interface to many external solvers. Since AMPL cannot generate the reformulation Q of P iteratively independently of the problem structure, we employed a C++ program that reads an AMPL output `.nl` file in flat form (92) and outputs the required reformulation as an AMPL-readable `.mod` file.

The `minlp_bb` solver (90) was found to be the MINLP solver that performs best when finding feasible points in nonconvex MINLPs (the comparison was carried out with the default-configured versions of `filMINT` (2) and `BonMin` (26)). The SQP solver of choice was `snopt` (65), found to be somewhat more reliable than `filtersqp` (59): on the analysed test set, `snopt` achieves, on average, better results at finding feasible solution in a short CPU time. All computational results have been obtained on an Intel Xeon 2.4 GHz with 8 GB RAM running Linux.

RECIPE rests on three configurable parameters: $k_{\max}$ (the maximum neighbourhood radius), L (the number of local searches starting in each neighbourhood) and the maximum allowed user CPU time (not including the time taken to complete the last local search). After some practical experimentation on a reduced subset of instances, we set $k_{\max} = 50$, $L = 15$ and the maximum CPU time to 10h. These parameters were left unchanged over the whole test set, yielding

good results without the need for fine-tuning.

8.3.1 MINLPLib

The MINLPLib (30) is a collection of Mixed Integer Nonlinear Programming models which can be searched and downloaded for free. Statistics for the instances in the MINLPLib are available from:

`http://www.gamsworld.org/minlp/minlplib/minlpstat.htm`.

The instance library is available at:

`http://www.gamsworld.org/minlp/minlplib.htm`.

The MINLPLib is distributed in GAMS (28) format, so we employed an automatic translator to cast the files in AMPL format.

At the time of downloading (Feb. 2008), the MINLPLib consisted of 265 MINLP instances contributed by the scientific and industrial OR community. These were all tested with the RECIPE algorithm implementation described above. We had 20 unsuccessful runs due to some AMPL-related errors (the model contained some unusual AMPL operator not implemented by some of the solvers or reformulators employed in RECIPE). The instances leading to AMPL-related failure were:

`blendgap`, `dosemin2d`, `dosemin3d`, `fuzzy`, `hda`, `meanvarxsc`, `pb302035`, `pb302055`, `pb302075`, `pb302095`, `pb351535`, `pb351555`, `pb351575`, `pb351595`, `water3`, `waterful2`, `watersbp`, `waters`, `watersym1`, `watersym2`.

The performance of RECIPE was evaluated on the 245 runs that came to completion. The results are given in Tables 8.1, 8.2 (solved instances) and 8.3 (unsolved instances). Table 8.1 lists results where the best solution found by RECIPE was different by at least 0.1% from that listed in MINLPLib. The first column contains the instance name, the second contains the value f^* of the objective function found by the RECIPE algorithm and the third the corresponding CPU usage measured in seconds of user time; the fourth contains the value $\bar{f}$ of the objective function reported in the official MINLPLib table and the fifth contains the name of corresponding GAMS solver that found the solution. The fourth and fifth columns were generated using the data available from the MINLPLib website, updated with some improved results obtained by AlphaECP (137) that are not yet listed on the MINLPLib web page. Table 8.2 lists instance names where the best values found by RECIPE and listed in MINLPLib are identical.

Instance	RECIPE		Known solution	
	f^*	*CPU*	$\bar{f}$	Solver
`csched2a`	**-165398.701331**	75.957500	-160037.701300	BonMin
`eniplac`	-131926.917119	113.761000	**-132117.083000**	SBB+CONOPT
`ex1233`	160448.638212	3.426480	**155010.671300**	SBB+CONOPT

Instance	RECIPE		Known solution	
	f^*	*CPU*	$\bar{f}$	Solver
ex1243	118489.866394	5.329190	**83402.506400**	BARON
ex1244	211313.560000	7.548850	**82042.905200**	SBB+CONOPT
ex1265a	15.100000	9.644530	**10.300000**	BARON
ex3	**-53.990210**	1.813720	68.009700	SBB+CONOPT
ex3pb	**-53.990210**	1.790730	68.009700	SBB+CONOPT
fo7_2	22.833307	23.710400	**17.748900**	AlphaECP
fo7	24.311289	25.423100	**20.729700**	AlphaECP
fo9	38.500000	46.296000	**23.426300**	AlphaECP
fuel	17175.000000	1.161820	**8566.119000**	SBB+CONOPT
gear4	1.968201	9.524550	**1.643400**	SBB+CONOPT2
lop97ic	4814.451760	3047.110000	**4284.590500**	-
lop97icx	**4222.273030**	1291.510000	4326.147700	SBB+CONOPT
m7	220.530055	17.275400	**106.756900**	AlphaECP
minlphix	**209.149396***	4.849260	316.692700	SBB+snopt
nuclear14b	**-1.119531**	7479.710000	-1.113500	SBB+CONOPT
nuclear24b	**-1.119531**	7483.530000	-1.113500	SBB+CONOPT
nuclear25	**-1.120175**	1329.530000	-1.118600	SBB+CONOPT
nuclearva	-1.008822	167.102000	**-1.012500**	SBB+CONOPT2+snopt
nuclearvb	-1.028122	155.513000	**-1.030400**	SBB+CONOPT2+snopt
nuclearvc	**-1.000754**	176.075000	-0.998300	SBB+CONOPT2+snopt
nuclearvd	**-1.033279**	202.416000	-1.028500	SBB+CONOPT2+snopt
nuclearve	-1.031364	193.764000	**-1.035100**	SBB+CONOPT2+snopt
nuclearvf	**-1.020808**	200.154000	-1.017700	SBB+CONOPT2+snopt
nvs02	**5.964189**	1.925710	5.984600	SBB+CONOPT3
nvs05	28.433982	4.215360	**5.470900**	SBB+CONOPT3
nvs14	**-40358.114150**	2.070690	-40153.723700	SBB+CONOPT3
nvs22	28.947660	4.849260	**6.058200**	SBB+CONOPT3
o7_2	125.907318	23.262500	**116.945900**	AlphaECP
o7	160.218617	24.267300	**131.649300**	AlphaECP
oil	-0.006926	389.266000	**-0.932500**	SBB+CONOPT(fail)
product	-1971.757941	2952.160000	**-2142.948100**	DICOPT+CONOPT3/CPLEX
st_e13	2.236072	0.548916	**2.000000**	BARON
st_e40	52.970520	0.930858	**30.414200**	BARON
stockcycle	120637.913333	17403.200000	**119948.688300**	SBB+CONOPT
super3t	-0.674621	38185.500000	**-0.685965**	SBB+CONOPT
synheat	186347.748738	3.534460	**154997.334900**	SBB+CONOPT
tln7	19.300000	1000.640000	**15.000000**	BARON
risk2b	$-\infty^*$	45.559100	-55.876100	SBB+CONOPT3
risk2bpb	$-\infty^*$	48.057700	-55.876100	SBB+CONOPT3

Table 8.1: Computational results on MINLPLib. Values denoted by * mark instances with unbounded values in the solution.

alan	ex1224	gbd	nvs06	parallel	st_e32	tln2
batchdes	ex1225	gear2	nvs07	prob02	st_e36	tln4
batch	ex1226	gear3	nvs08	prob03	st_e38	tln5
cecil_13	ex1252a	gear	nvs09	prob10	st_miqp1	tln6
contvar	ex1252	gkocis	nvs10	procsel	st_miqp2	tloss
csched1a	ex1263a	hmittelman	nvs11	pump	st_miqp3	tls2
csched1	ex1263	johnall	nvs12	qap	st_miqp4	util
csched2	ex1264a	m3	nvs13	ravem	st_miqp5	
du-opt5	ex1264	m6	nvs15	ravempb	st_test1	
du-opt	ex1265	meanvarx	nvs16	sep1	st_test2	
enpro48	ex1266a	nuclear14a	nvs17	space25a	st_test3	
enpro48pb	ex1266	nuclear14	nvs18	space25	st_test4	
enpro56	ex4	nuclear24a	nvs19	spectra2	st_test6	
enpro56pb	fac1	nuclear24	nvs20	spring	st_test8	
ex1221	fac2	nuclear25a	nvs21	st_e14	st_testgr1	
ex1222	fac3	nuclear25b	nvs23	st_e15	st_testph4	
ex1223a	feedtray2	nvs01	nvs24	st_e27	synthes1	

ex1223b	feedtray	nvs04	oaer	st_e29	synthes2	
ex1223	gastrans	nvs03	oil2	st_e31	synthes3	

Table 8.2: Instances for which RECIPE's optima are the same as those reported in the MINLPLib.

8.3.2 Optimality

RECIPE found feasible solutions for 163 instances out of 245 (66%). Relative to this reduced instance set, it found the best known solution for 121 instances (74%), gave evidence of the unboundedness of 3 instances (1%), and improved the best known objective value for 12 instances (7%). In the other cases it found a local optimum that was worse than the best known solution.

Improved solutions were found for the following instances:

csched2a:	$f^* = -165398.701331$	(best known solution: -160037.701300)
ex3:	$f^* = -53.990210$	(best known solution: 68.009700)
ex3pb:	$f^* = -53.990210$	(best known solution: 68.009700)
lop97icx:	$f^* = 4222.273030$	(best known solution: 4326.147700)
minlphix:	$f^* = 209.149396$	(best known solution: 316.692700)
nuclear14b:	$f^* = -1.119531$	(best known solution: -1.113500)
nuclear24b:	$f^* = -1.119531$	(best known solution: -1.113500)
nuclear25:	$f^* = -1.120175$	(best known solution: -1.118600)
nuclearvc:	$f^* = -1.000754$	(best known solution: -0.998300)
nuclearvd:	$f^* = -1.033279$	(best known solution: -1.028500)
nuclearvf:	$f^* = -1.020808$	(best known solution: -1.017700)
nvs02:	$f^* = 5.964189$	(best known solution: 5.984600)
nvs14:	$f^* = -40358.114150$	(best known solution: -40153.723700)
risk2b:	$f^* = -\infty$	(best known solution: -55.876100)
risk2bpb:	$f^* = -\infty$	(best known solution: -55.876100).

All new best solutions were double-checked for constraint, bounds and integrality feasibility besides the verifications provided by the local solvers, and were all found to be integral feasible; 11 out of 12 were constraint/bound feasible to within a 10^{-5} absolute tolerance, and 1 (csched2a) to within 10^{-2}. The 3 instances marked by * in Table 8.1 (minlphix, risk2b, risk2bpb) gave solutions x^* with some of the components at values in excess of 10^{18}. Since minlphix minimizes a fractional objective function and there are no upper bounds on several of the problem variables, the optimum is attained when the variables appearing in the denominators tend towards $+\infty$. We solved risk2b and risk2bpb several times, setting increasing upper bounds to the unbounded variables: this yielded decreasing values of the objective function, suggesting that these instances are really unbounded (hence the $-\infty$ in Table 8.1).

4stufen	eg_int_s	fo8_ar4_1	m7_ar4_1	nuclear10a	o9_ar4_1	super3	waste
beuster	elf	fo8_ar5_1	m7_ar5_1	nuclear10b	ortez	tln12	water4
deb10	fo7_ar2_1	fo9_ar2_1	mbtd	nuclear49	product2	tls12	waterx
deb6	fo7_ar25_1	fo9_ar25_1	no7_ar2_1	nuclear49a	qapw	tls4	waterz
deb7	fo7_ar3_1	fo9_ar3_1	no7_ar25_1	nuclear49b	saa_2	tls5	windfac
deb8	fo7_ar4_1	fo9_ar4_1	no7_ar3_1	o7_ar2_1	space960	tls6	
deb9	fo7_ar5_1	fo9_ar5_1	no7_ar4_1	o7_ar25_1	st_e35	tls7	
detf1	fo8	gasnet	no7_ar5_1	o7_ar3_1	st_test5	tltr	
eg_all_s	fo8_ar2_1	m7_ar2_1	nous1	o7_ar4_1	st_testgr3	uselinear	
eg_disc2_s	fo8_ar25_1	m7_ar25_1	nous2	o7_ar5_1	super1	var_con10	
eg_disc_s	fo8_ar3_1	m7_ar3_1	nuclear104	o8_ar4_1	super2	var_con5	

Table 8.3: Instances unsolved by RECIPE.

On 82 instances out of 245 listed in Table 8.3, RECIPE failed to find any local optimum within the allotted time limit. Most of these failures are due to the difficulty of the continuous relaxation of the MINLPs: there are several instances where the SQP method (`snopt`) does not manage to find a feasible starting point, and in these cases the convex MINLP solver (`minlp_bb`) also fails. On a smaller number of instances, `minlp_bb` is not able to find integral feasible solutions even though constraint feasible solutions are provided by `snopt`.

8.3.3 Reliability

One interesting feature of RECIPE is its reliability: in its default configuration it managed to find solution with better or equal quality than those reported in the MINLPLib on 136 instances over 245 (55%) and at least a feasible point in a further 11% of the cases. On the same set of test instances, the closest competitor is SBB+CONOPT, which matches or surpasses the best solutions in MINLPLib in 37% of the cases, followed by BARON with 15% and by AlphaECP with 14%. These percentages were compiled in June 2008 by looking at:

`http://www.gamsworld.org/minlp/minlplib/points.htm`.

8.3.4 Speed

The total time taken for solving the whole MINLPLib (including the unsolved instances, where the VNS algorithm terminates after exploring the neighbourhoods up to $k_{\max}$ or when reaching the 10 hours time limit, whichever comes first) is roughly 4 days and 19 hours of user CPU time. RECIPE's speed is very competitive with that of sBB approaches: tests conduced using the *ooOPS* solver (91; 92; 100) as well as BARON on some complex MINLPs showed that sBB methods may take a long time to converge. Naturally, the trade-off for this speed is the lack of an optimality guarantee.

Chapter 9

Computational Experiments on the TDSPP

Throughout this part, we tested our methods on common benchmark instances taken from the literature, so that we could compare with other works. We now revert our attention back to the time-dependent shortest paths problem. In this chapter we test the most efficient algorithms proposed so far on shortest path instances, using a combination of real-world and synthetic data.

The rest of this chapter is organized as follows. In Section 9.1 we discuss the input data for our numerical tests. In Section 9.2 we test the linear formulation of the TDSPP with a Branch-and-Bound algorithm which uses the branching rules proposed in Chapter 7. In Section 9.3 we apply the VNS algorithm of Chapter 8 to the MINLP formulation of the TDSPP.

9.1 Input data

For our numerical experiments, we extracted several subnetworks of increasing size from the road network of Rome. The original graph is available from the web page of the 9^{th} DIMACS Challenge – Shortest Paths:

`http://www.dis.uniroma1.it/~challenge9/.`

There are 3353 vertices and 8870 oriented arcs in the road network; however, our formulation $(GTDSPP)$ contains one (integer) variable for each arc and two variables for each node of the graph, as well as additional variables needed to model arc costs. Therefore, to keep computational times down to acceptable levels, we considered subnetworks extracted from this graph instead of the full size graph. In particular, we tested our methods on networks with the characteristics reported in Table 9.1. Due to the large number of variables and constraints that arise in the formulations (both linear and nonlinear), we were not able to carry out computational experiments in reasonable time on instances with more arcs and nodes than those reported in Table 9.1.

NAME	NODES	ARCS
R-8	8	14
R-30	30	62
R-60	60	144
R-120	120	296
R-200	200	504

Table 9.1: Size of the test instances.

Time-dependent costs were generated following some simple guidelines. For piecewise linear cost functions, we considered 3 breakpoints on each arc. Although this may seem like a small number, the trouble lies in the fact that each breakpoint introduces additional variables. Moreover, our approach is based on the consideration that, instead of assigning to each arc a cost profile for a full day and using a nonzero departure times for each shortest path computation, we can consider the departure time to be always zero and "shift" the cost profiles accordingly. That is, the 3 breakpoints can be considered as the first 3 breakpoints on the cost profile after the desired departure time. If the time horizon covered by the 3 breakpoints is sufficiently large, this approach is equivalent to considering the full cost profiles, but requires significantly less effort. Breakpoints were positioned randomly within a time horizon of 3 hours, and their value for arc (u, v) was selected uniformly at random in $[\lambda(u, v), 20\lambda(u, v)]$, where, as usual, $\lambda(u, v)$ is the travelling time over (u, v) in traffic free situation. Note that we did not enforce the FIFO property when generating arc costs.

For the nonlinear model, we used the cost functions proposed in Section 1.2.2 involving a summation of Gaussians. We set the number of cost perturbations to two, i.e. each cost function is of the form

$$c(i, j, \tau) = c_{ij} + \sum_{k=1}^{2} a_k e^{-\frac{(\tau-\mu_k)^2}{2\sigma_k^2}},$$

We used the same idea of "shifting" cost profiles described above. Therefore, the departure time of each shortest path is considered to be always zero, and the cost perturbations that correspond to traffic jams are centered within a time horizon of 3 hours. The mean μ_k for each gaussian is picked uniformly at random in the time interval between 0 and 3 hours (rounding to the nearest second), whereas the standard deviation σ_k is always 900 (we remark that time is expressed in seconds). Each Gaussian function on arc (u, v) is multiplied by a factor a_k that is chosen uniformly between $[\lambda(u, v), 20\lambda(u, v)]$.

9.2 Numerical experiments with the linear formulation

In this section we discuss the computational experiments carried out on the MILP formulation for the TDSPP. First, we give more details on the formulation; then, we analyse the computational results.

9.2.1 Formulation

Consider the formulation $(GTDSPP)$ presented in Section 1.3:

$$\left.\begin{array}{rrcl} \min & \tau_t & & \\ \forall v \in V & \sum_{(i,j)\in A} m_{ij}^v x_{ij} & = & b_v \\ \forall v \in V & \tau_v & \leq & d_v \\ \forall (i,j) \in A & x_{ij}(d_i + c(i,j,d_i)) & \leq & \tau_j \\ \forall (i,j) \in A & x_{ij} & \in & \{0,1\} \\ \forall v \in V & \tau_i & \geq & 0 \end{array}\right\} (GTDSPP)$$

In this section, we consider $c(i,j,d_i)$ to be a piecewise linear function for all arcs $(i,j) \in A$. We are given a set of breakpoint positions $brp_{ij}^k, k = 1,\ldots,h$ and a set of corresponding breakpoint values $brv_{ij}^k, k = 1,\ldots,h$ for each arc (i,j). We add binary variables δ_{ij}^k that select which piece of the piecewise linear function on (i,j) is active for a given departure time τ_i from node i. Note that, for simplicity, we considered each arc cost function expressed as a sum between a fixed (static) cost c_{ij} and a piecewise linear function, described as above. This way, for all time instants following the last breakpoint brp_{ij}^h, the cost of an arc is given by its fixed cost only. Therefore, the formulation becomes:

$$\left.\begin{array}{rrcl} \min & \tau_t & & \\ \forall v \in V & \sum_{(i,j)\in A} m_{ij}^v x_{ij} & = & b_v \\ \forall v \in V & \tau_v & \leq & d_v \\ \forall (i,j) \in A & x_{ij}(d_i + c_{ij} \sum_{k=1}^{h-1} \delta_{ij}^k (\frac{d_i - brp_{ij}^k}{brp_{ij}^{k+1} - brp_{ij}^k}(brv_{ij}^{k+1} - brv_{ij}^k) + brv_{ij}^k)) & \leq & \tau_j \\ \forall (i,j) \in A & \sum_{k=1}^{h-1} \delta_{ij}^k brp_{ij}^k + \delta_{ij}^h M & \geq & d_i \\ \forall (i,j) \in A & \sum_{k=2}^{h} \delta_{ij}^k brp_{ij}^{k-1} & \leq & d_i \\ \forall (i,j) \in A & \sum_{k=1}^{h} \delta_{ij}^k & = & 1 \\ \forall (i,j) \in A & x_{ij} & \in & \{0,1\} \\ \forall (i,j) \in A & \delta_{ij}^k & \in & \{0,1\} \\ \forall v \in V & \tau_i & \geq & 0. \end{array}\right\}$$

The third constraint is the complicating one: it involves the product between a binary variable and a continuous variable, and between two binary variables and a continuous variable. All these products can be formulated exactly with linear terms, introducing additional variables (see e.g. (94)).

The resulting MILP has a weak LP relaxation, due to the product reformulations which are known to be weak. Therefore, solving large instances to proven optimality is a very difficult task. However, our aim is to compare on this class of problems traditional branching rules with the methods studied in this thesis. We note that, if no path between the source node and the target node exists, then this is detected at root node, since the LP relaxation is infeasible because of the flow conservation constraints.

9.2.2 Computational results

In Table 9.2 we report the number of variables and of constraints for all test instances with the MILP formulation described Section 9.2.1, *before* preprocessing. Although the number of variables and constraints is large, the constraint matrix is sparse, hence the number of nonzeroes is small; this is because the flow conservation constraints have only two nonzeroes per row. Note that the size of the formulation grows rapidly with the size of the network; a great deal of this growth is due to the complexity of modeling the arc cost functions, which requires several variables and constraints for each arc of the network.

NAME	# VARIABLES	# CONSTRAINTS
R-8	142	282
R-30	618	1238
R-60	1342	2780
R-120	2734	5666
R-200	4396	9976

Table 9.2: Number of variables and of constraints of the MILP formulation for the TDSPP.

The setup for numerical experiments is as follows: we compare the CGD and SD branching algorithms discussed in Chapter 7 on the instances described in Section 9.1. For each instance, we selected 100 unique source/target pairs at random, and used our implementations of CGD and SD. For the smallest instance (R-8), instead of selecting 100 source/target pairs at random, we tested all possible source/target pairs. We allow Cplex's cutting planes and preprocessing at root node, with default parameters; time limit is set to two hours, whereas there is no limit on the number of nodes. Since the value of the optimum is not known in advance, heuristics were *not* disabled, in contrast to the experiments in Chapter 7.

We report, for each instance, average values over all the shortest paths computations. Source/destination pairs that are solved in less than two nodes by both methods are not considered when computing the averages. We report the following values (in the order in which they appear as columns):

- percentage of instances solved to optimality by SD within the two hours time limit;
- percentage of instances solved to optimality by CGD within the two hours time limit;
- average number of nodes enumerated by SD on instances which are solved by both methods;
- average CPU time (in seconds) of SD on instances which are solved by both methods;
- average number of nodes enumerated by CGD on instances which are solved by both methods;
- average CPU time (in seconds) of CGD on instances which are solved by both methods;
- average number of nodes enumerated by SD on instances which are unsolved by either method;
- average relative integrality gap left by SD on instances unsolved by either method, computed as $(ub-lb)/ub$ where ub is the best known upper bound (i.e. primal feasible solution) and lb the best known lower bound after two hours;
- average number of nodes enumerated by CGD on instances which are unsolved by either method;
- average relative integrality gap left by CGD on instances unsolved by either method, computed as $(ub - lb)/ub$ where ub is the best known upper bound and lb the best known lower bound after two hours.

All averages are geometric, computed as in Section 7.4. Note that, since we do not known the optimal solution a priori, we record the amount of relative integrality gap *left*, i.e. not closed; therefore, optimality is attained when this value becomes 0%. On all instances, Cplex's heuristics were able to find at least one feasible solution. The first feasible solution is typically discovered at root node in almost all cases.

Table 9.3 shows that CGD consistently performs better than SD on average. On the smallest road network (R-8), all shortest path computations are solved to optimality at root node; therefore, CGD and SD perform equally. As the size of the road network increases, which corresponds to a larger number of variables and constraints, CGD closes a larger amount of gap per node. Instances which are solved by both methods require fewer nodes and less CPU time on average when employing our improved branching strategy CGD; similarly, on

	Num. solved		Solved				Unsolved			
			SD		CGD		SD		CGD	
	SD	CGD	nodes	time	nodes	time	nodes	gap	nodes	gap
R-8	100%	100%	0	0.02	0	0.02	-	-	-	-
R-30	98%	93%	2508	345.95	80	34.15	-	-	-	-
R-60	55%	60%	845	82.95	74	22.75	57133	87.7%	30792	70.4%
R-120	40%	40%	317	62.39	53	48.50	23991	83.6%	13405	75.8%
R-200	23%	42%	1349	701.08	378	433.46	8599	97.4%	4385	93.7%

Table 9.3: Comparison of the SD and CGD branching algorithms on the MILP formulation for the time-dependent shortest paths problem.

instances which are unsolved by either method we close a larger integrality gap, even though we enumerate fewer nodes in the 2 hours time limit. Although it may look counterintuitive that the number of nodes (and CPU time for unsolved instances) decreases as the size of the instance increases, this depends on the way the average values are computed. In fact, results are computed on instances that are solved by both methods, and on instances that are unsolved by either method. As the size of the instances grows, only very easy shortest path computations are carried out to optimality; therefore, there is a decrease in the number of nodes required on average on solved instances, but the number of such instances also decreases (which can be seen in the second and third column). Similarly, we enumerate fewer nodes in two hours on the unsolved instances, because the LP relaxation becomes more expensive. We note that, for moderate size instances already, not all shortest path computations can be carried out to optimality within the time limit; this is due to the large number of variables and constaints, and to the weakness of the LP relaxation, which makes increasing the lower bound very difficult. In order to deal with full city-sized road networks, it would probably be necessary to give up some accuracy in the network modeling, so as to have a manageable number of variables and constraints. However, the most effective branching strategy proposed in Chapter 7 shows an improvement over traditional branching rules on shortest path instances too, assessing its practical usefulness.

9.3 Numerical experiments with the nonlinear formulation

In this section we discuss the computational experiments carried out on the MINLP formulation for the TDSPP. As for the linear case, we give more details on the formulation; then, we discuss the modifications that we applied to the RECIPE algorithm (Chapter 8) to increase its performance on this particular kind of problems. Finally, we present computational results.

9.3.1 Formulation

We tested our VNS-based heuristic algorithm for MINLPs on the following mathematical formulation:

$$\left.\begin{array}{rll} \min & \tau_t & \\ \forall v \in V & \sum_{(i,j)\in A} m_{ij}^v x_{ij} & = b_v \\ \forall v \in V & \tau_v & \leq d_v \\ \forall (i,j) \in A & x_{ij}(d_i + c_{ij} + \sum_{k=1}^{2} a_k e^{-\frac{(d_i-\mu_k)^2}{2\sigma_k^2}}) & \leq \tau_j \\ \forall (i,j) \in A & x_{ij} & \in \{0,1\} \\ \forall v \in V & \tau_i & \geq 0. \end{array}\right\}$$

In the third constraint, we reformulated the product between x_{ij} and d_i introducing an additional variable for each product and additional linear constraints (94).

9.3.2 Modifications to RECIPE

The RECIPE algorithm is a general purpose VNS-based heuristic for nonconvex MINLPs. However, in this section we are focusing on a particular class of problems, that share a common structure. Therefore, we slightly modified RECIPE to increase its performance on the problem at hand. The modifications affect the initialization phase of the algorithm. Recall that RECIPE is basically a multistart algorithm until a first feasible solution is found; only then, the main part of the algorithm enters into play. In the general purpose algorithm described in Chapter 8, the initialization phase alternates between a continuous NLP solver and a (convex) MINLP solver, with different starting points, until the first feasible solution is found. For the shortest paths problem, we introduced a more complex initialization phase, that always provides a feasible solution (if one exists) by applying a simple heuristic.

The new initialization phase works as follows. First, we compute the shortest path between source and target node on the graph with static costs c_{ij}. This is done very quickly because it only requires solving a linear program, and provides a feasible path p between the source and the target. Then, we set the departure time from the source node to 0, and follow the path p until we reach the target. For each arc (i,j) that we encounter on the path, we have already computed a feasible departure time d_i from i. We calculate the time-dependent cost $c(i,j,d_i)$ of arc (i,j) at time d_i, and set τ_j and d_j equal to $d_i + c(i,j,d_i)$. This yields a feasible assignment of all the variables, and can be done in linear time in the number of arcs. Note that this step can be carried out regardless of the form of $c(i,j,d_i)$.

The modified initialization phase provides an initial feasible solution in a very short time. We employ Cplex to solve the linear program that yields the initial feasible path p; in our experiments, this always required less than a second

of CPU time. Although the solution that we compute through this heuristic may be far from the optimum, it still provides a starting point for our VNS-based algorithm. An additional advantage is that we are able to detect immediately if there is no path between the source and the destination node, i.e. if no feasible solution exists.

9.3.3 Computational results

Numerical experiments were run as follows: for each instance described in Section 9.1, we generated 100 distinct source/destination pairs at random, and employed the RECIPE algorithm that is discussed in Chapter 8 with the modifications presented in Section 9.3.2. For the smallest instance (R-8), instead of selecting 100 source/target pairs at random, we tested all possible source/target pairs.

We ran some preliminary tests with different solvers employed on this formulation within RECIPE (`minlp_bb` and `bonmin` (26)), as well as the exact Branch-and-Bound solver `couenne` for nonconvex MINLPs (see (23; 22)). All these solvers experienced severe numerical difficulties due to the summation of Gaussian functions, which turned out to be very difficult to treat: in several cases, the continuous relaxation of the problem is not solved to feasibility by the NLP solvers employed within `minlp_bb`, `bonmin` and `couenne`, even though a feasible solution exists. Therefore, we decided to replace each Gaussian function $\mathcal{G}(\mu_k, \sigma_k)$ with its Taylor series up to the fourth term, centered at μ_k, for all time instants less than $3\sigma_k$ away from the mean. The contribution of each Gaussian was considered to be zero for $\tau \in \mathcal{T} : |\tau - \mu_k| \geq 3\sigma_k$. To truncate the power series, we used a binary variable for each Gaussian term, which indicates whether the function is active or not. Note that the feasibility heuristic described in Section 9.3.2 can still be applied: we compute a feasible assignment of the binary variables that determine whether a Gaussian function on arc (i, j) is active or not (depending on the departure time d_i from node i), then we compute the cost of the arc defined through the Taylor series.

In Table 9.4 we report the number of variables and of constraints for all test instances with the MINLP formulation described above, *before* preprocessing. Again, the flow conservation constraints have only two nonzeroes per row, therefore part of the constraint matrix is very sparse. We were not able to solve instances larger than R-60 in a reasonable amount of time; in particular, the convex MINLP solver `minlp_bb` employed during the local search phase by our RECIPE algorithm is not able to provide a solution within the 3 hours time limit for instances larger than R-60. The same applies to `couenne`, whose primal heuristics fail due to the size of the problem, and no feasible solution is found within the time limit.

Since the NLP relaxations encountered during the solution process are expensive, and we did not want to invest too much time on each shortest path

NAME	# VARIABLES	# CONSTRAINTS
R-8	142	142
R-30	308	308
R-60	696	696

Table 9.4: Number of variables and of constraints of the MINLP formulation for the TDSPP.

computation, we used different parameters than those reported in Chapter 8: we set $k_{\max} = 30$, $L = 5$, and 3 hours as maximum CPU time. We compare our results with the solution obtained with the nonconvex MINLP solver `couenne`, setting a 3 hours time limit. We report the following statistics, listed in the order in which they appear as columns:

- percentage of instances on which RECIPE finds a better feasible solution with respect to `couenne` (with a relative improvement of at least 1% in the value of the objective function);
- percentage of instances on which `couenne` finds a better feasible solution with respect to RECIPE (with a relative improvement of at least 1% in the value of the objective function);
- average relative integrality gap left by RECIPE, computed as $(ub - lb)/ub$ where ub is the best feasible solution found and lb the best known lower bound provided by `couenne` after the two hours time limit;
- average relative integrality gap left by `couenne`, computed as $(ub - lb)/ub$ where ub is the best feasible solution found and lb the best known lower bound after the two hours time limit;
- average relative improvement of the solution computed by RECIPE with respect to the best solution provided by `couenne`, computed as:

$$|f^*_{RECIPE} - f^*_{\texttt{couenne}}|/f^*_{RECIPE};$$

- average CPU time required by RECIPE (in seconds);
- average CPU time required by `couenne` (in seconds).

All averages are geometric.

The results reported in Table 9.5 assess the reliability of our VNS-based heuristic with respect to exact solution methods such as the one implemented by `couenne`. The feasible solution provided by RECIPE is always at least as good as the one provided by `couenne` in all runs. On the smallest instance (R-8), RECIPE and `couenne` always find the same solution, which is proven to be optimal. RECIPE takes slightly longer before termination because it starts several

	# BETTER SOL.		INTEGRALITY GAP		IMPR.	CPU TIME	
	RECIPE	couenne	RECIPE	couenne		RECIPE	couenne
R-8	0%	0%	0%	0.0%	0.0%	9.8	2.0
R-30	53%	0%	82.3%	88.4%	2.3%	6189.2	10848.6
R-60	80%	0%	99.5%	100.0%	28.5%	8514.0	10788.4

Table 9.5: Comparison of RECIPE and couenne on the MINLP formulation for the time-dependent shortest paths problem.

local searches, whereas couenne is able to prove optimality in a very short time. Overall, on such a small instance using an exact method such as couenne seems like a better option, because we have a guarantee of optimality and running times are short. However, if we increase the size of the underlying road network, then RECIPE becomes more appealing with respect to couenne: on average, the solution quality increases by a factor that becomes larger as the size of the instances grows. Correspondigly, the amount of integrality *not* closed decreases. Note that for the R-60 instance the amount of integrality gap left is almost 100%, because on most instances couenne is not able to increase the lower bound to a value larger than zero; but the improvement in the objective value of the solution found by RECIPE with respect to couenne is significant. Moreover, RECIPE is fast: on several instances, an application of RECIPE terminates before the 3 hours time limit, whereas couenne hits the time limit on almost all instances (which is shown by an average running time close to 3 hours). Summarizing, our heuristic algorithm performs on average very well on these shortest path instances, finding better solutions than the exact Branch-and-Bound solver couenne within the allotted time frame; the only exception is given by very small instances, where RECIPE and couenne always find the same optimal solution, but couenne is slightly faster. Unfortunately, we were not able to apply RECIPE to larger instances because of the failure of the local solvers when dealing with too many variables and constraints, but we have no reasons to believe that RECIPE would not scale well if the local solvers were able to deal with larger problem sizes.

Part III

Conclusions and Bibliography

Chapter 10

Summary and Future Research

10.1 Summary

We considered the problem of finding the shortest path between two nodes on a large-scale time-dependent graph, which has many interesting practical applications. The problem is theoretically solved in an efficient way by Dijkstra's algorithm under the FIFO property, but there are many real-time applications where employing Dijkstra's algorithm would take too much time. Thus, we analysed speedup techniques for this algorithm. Moreover, we proposed a mathematical programming formulation for the point-to-point shortest path problem, starting from a classical formulation for static graphs and modifying it in order to take into account time-dependency in non-FIFO networks and possibly nonlinear time-dependent cost functions. We analyzed the resulting formulation, which is a MILP if the arc cost functions are linear or piecewise linear, whereas it is a MINLP if they are arbitrary nonlinear functions. We studied algorithms for both classed of problems, and tested them on both benchmark instances taken from the literature and shortest path instances.

First, we reviewed existing speedup techniques, both for static graphs and for time-dependent graphs. This allowed us to underline that the fastest algorithms typically rely on hierarchical approaches, sometimes combined with goal directed search. However, hierarchical methods are inherently bidirectional, and bidirectional search cannot be directly applied on time-dependent graphs. Therefore, we tried to overcome this problem. We discussed a method based on defining small node sets with an approximation guarantee; Dijkstra searches are constrained to explore only nodes in these precomputed sets, hence the number of touched nodes decreases. We analyzed advantages and drawbacks of this approach. We developed a general framework for bidirectional search on time-dependent graphs; the main idea is to run a time-dependent forward search, and a time-independent backward search whose purpose is to bound the set of nodes explored by the forward search. Following this approach, we are able to improve the efficacy of the ALT algorithm, which is a

clever application of the well known A^* algorithm to road networks; part of this improvement is due to the strengthening of the lower bounds to distances within the graph which are used throughout the algorithm. We proved correctness of the proposed method, and also proposed several enhancements. As a consequence, we were able to apply a two-levels hierarchical approach on time-dependent graphs. This is based on the idea of selecting a small subnetwork which contains important arcs; then, most of the calculations are carried out on the subnetwork, so that fewer nodes have to be explored. In analogy with the real world, this can be viewed as the motorway network, which is a subset of the original road network: when planning the route between two sufficiently distant points, local roads are considered only in a small radius centered on the departure and the destination point, but most of the path relies on motorway segments. From a theoretical point of view, our two-levels hierarchical approach could be extended to a multi-level hierarchy. However, our computational experiments, as well as numerical experiences reported in the literature, suggest that multi-level hierarchies do not work well for time-dependent graphs as they do for static graphs. Therefore, we only considered at most two levels.

One of the advantages of our method is its straightforward extension to dynamic scenarios, that is, scenarios where the cost functions on arcs are not fixed, but can be updated from time to time. This is a practical problem which has been previously described in the literature for the static case, but which had not been tackled in the time-dependent case. It turns out that we only need a small computational overhead to restore optimality of the hierarchy whenever some cost functions are updated. The amount of necessary work depends on the number and the position of the updated arcs: our algorithm benefits from spatial locality; therefore, if several contiguous arcs have their cost changed, we can run the update in an efficient way. We remark that, for real world applications, this is often the case: traffic jams typically extend over several adjacent road segments.

We provided extensive computational experiments to show the effectiveness of our approach on real world data. We tested the algorithm on two different road networks: the road network of Western Europe with generated time-dependent data, which is commonly used as a benchmark in the literature, and the road network of France with real world time-dependent data. We analysed the performance of our algorithm taking into account different criteria, and comparing with other existing methods. The main improvement of our approach with respect to the existing algorithms is the significant reduction (two orders of magnitude, if comparing to Dijkstra's algorithm or unidirectional ALT) of the average number of settled nodes for a path computation, which leads to average query times of a few milliseconds. Only the recently developed SHARC algorithm is faster than our method; however, SHARC cannot deal with the dynamic scenario. Moreover, our method can also compute approximated solu-

tions, with approximation guarantee defined at query time; if we are willing to accept a maximum approximation of 5%, then our algorithm is faster than SHARC. Optimality of the hierarchy after modifications in the cost functions can be restored very quickly, if the algorithm is parameterized to do so; indeed, there is a trade off between query speed and update speed, which depends on the values of the parameters for the preprocessing phase. We analysed the algorithm performance for several values of these parameters, so as to provide enough information to strike a good balance for real world applications.

We described a real world application: we integrated an implementation of our algorithm in C++ within an existing industrial platform which collects, gathers and treats real-time traffic information and traffic forecasts. Our aim was to provide a path computation service for the website of the Mediamobile company, which deals with traffic information and, as such, required an algorithm capable of answering several shortest path queries per second taking into account the most recent available traffic forecasts. To this end, we proposed a least squares algorithm to modify the arc cost functions, so as to fit the traffic forecasts as much as possible at each update.

Within the context of solving MILPs with Branch-and-Bound, we proposed two methods to generate good split disjunctions for branching: a heuristic approach that solves a quadratic optimization problem, and a mathematical formulation that models the problem of finding a split disjunction closing a large gap at the current node. Computational experiments show that both methods are able to close more gap with respect to the traditional branching rule, which consists in branching on a single variable with fractional value. We proposed a combination of the heuristic procedure based on a quadratic formulation with branching on single variables, and showed that the resulting branching algorithm, which we called CGD, is very effective in practice. Indeed, on a large set of test instances, CGD performs better than branching on single variables only, both in terms of size of the enumeration tree and in terms of computing time. Moreover, we are able to solve one very difficult instance, which is typically not solved by commercial software. We implemented a branching scheme based on the mathematical formulation that models the problem of finding a split disjunction closing a large gap, and we tested it on several heterogeneous instances. This method is indeed capable of closing more gap with respect to branching on single variables on the majority of test instances, but it requires considerably more time, which does not seem to be worth the extra effort. If compared to the heuristic procedure based on a quadratic formulation, this approach seems to work better on the difficult instances, but the heuristic procedure is significantly faster.

Finally, we described a heuristic approach to solving nonconvex MINLPs. Our method, called RECIPE, combines several existing exact, approximate and heuristic techniques: the global search phase is cohordinated by the VNS metaheuristic, whereas the local search employs both continuous NLP and convex

MINLP solvers, with a neighbourhood structure defined through hyperrectangles and local branching. This results in an algorithm that can successfully solve many difficult MINLPs without hand-tuned parameter configuration. Such a reliable solver would be particularly useful in industrial applications where the optimum quality is of relative importance and the optimization layer is hidden from user intervention and is therefore "just supposed to work". Over a large collection of benchmark instances taken from the literature, not only we find the best known solution for the majority of the test problems, but in some cases we improve over the best known objective value, while requiring very low computational times.

The mathematical programming formulation that we proposed for the time-dependent shortest paths problem on non-FIFO networks turned out to be very challenging for MILP and MINLP solvers; one of the difficulties is given by the large number of variables and constraints that arise in the formulation and are necessary to model arc costs. Computational experiments on shortest path instances showed that, on the MILP formulation, the branching rules discussed in this thesis achieved good results, cutting down the number of required nodes by a significant factor on average. CPU times also benefited from the proposed approach. Despite developing a specially tailored initialization phase for RECIPE on this particular kind of shortest paths instances, we could only solve the MINLP formulation on small networks; numerical troubles affect both our heuristic VNS-based algorithm and exact solvers for nonconvex MINLPs. On the instances that we were able to solve, RECIPE showed good results, finding near-optimal solutions for almost all instances in a short CPU time.

Summarizing, we reviewed the most successfull algorithms for the point-to-point shortest path problem on static road networks, which rely on two different approaches, often mixed together: goal directed search and hierarchical routing. Our aim was to apply the same techniques on time-dependent road networks. To do so, we improved an existing algorithm for goal directed search in the time-dependent case, and we developed a framework for bidirectional search, that allowed an easy generalization of hierarchical routing to time-dependent networks. Computational experiments confirmed the efficacy in practice of our ideas. Moreover, our algorithm is the first method which is able to deal in an efficient way with dynamic time-dependent scenarios, that is, applications where the time-dependent arc cost functions are not fixed and known a priori, while still yielding a speedup of more than two orders of magnitude with respect to Dijkstra's algorithm. We proposed a mathematical programming formulation for the TDSPP. We studied improvements for the widely used Branch-and-Bound algorithm for MILPs, resulting in an average reduction of the number of enumerated nodes by a factor of two and an excellent perfomance on some well known hard instances. These improvements rely on the selection of a branching decision which is different from the standard rule

of branching on a single variable. We presented a general purpose heuristic for nonconvex MINLPs, which combines several off-the-shelf components into an effective and fast solver for this difficult class of problems. We tested the proposed approach on the mathematical formulation for the TDSPP, analyzing the computational results.

10.2 Future research

Even if the speed and versatility showed by the shortest paths algorithm proposed in this thesis should be sufficient for most practical applications, there is still much room for research in the field. For some applications, it would be desirable to have very fast query times and no additional overhead when changing the cost functions. Examples of this are route planners which use different cost functions depending on the vehicle type that is querying the path computing service. Although this seems an impossible challenge, it is still an interesting subject of research, maybe assuming some restrictions on the cost functions to simplify the problem. If the cost functions are similar, the SHARC algorithm discussed in Section 1.4.6 has showed promising result. Another interesting direction for future research is multicriteria optimization. Routing applications in general networks (not necessarily road networks) often have to deal with several objective functions that the user would like to minimize; e.g., travelling time and number of connections for railway routing, or travelling time and motorway fees for road networks. How to formalize this problem is unclear. Some approaches rely on finding all the Pareto optima, and let the user choose among them. However, computing all the Pareto optima is a difficult task, and could greatly benefit from speedup techniques. We believe that the techniques presented in this thesis could be used as a building brick for efficient algorithms in the multiobjective case. At the moment of finalizing this thesis, we are aware that an extension of the SHARC algorithm shows very good preliminary results (42). The greatest drawbacks of SHARC are its long preprocessing time and the capability of dealing with static scenarios only. It would be interesting to hybridize SHARC with the techniques proposed in this work, so as to be able to perform efficient multicriteria optimization on dynamic networks.

Concerning the computation of split disjunctions to be employed as branching decisions in MILPs, one of the main questions which still has to answered is: given a good disjunctions, is it more profitable to branch on it or to use it to generate a cutting plane? So far, the answer is not clear, and the question provides an interesting challenge to the integer programming community. Devising fast and efficient heuristics to generate strong disjunctions is another field which is currently studied by several research groups. It is important to note that at the moment it is not known how much benefit could be brought by a "perfect" branching strategy: for cutting planes, some studies have underlined

that there is still a large gap between the performance obtained by commonly implemented cutting planes and the theoretically achievable maximum. But it is difficult to undertake the same study for branching decisions. It is known that a decrease of the number of nodes of at least a factor of two can be obtained by carefully choosing a branching disjunction; if this result could be achieved with a very small computational effort, branching on general disjunctions would become interesting for commercial solvers as well. We believe that our work moves some steps in that direction.

Finally, heuristic approaches to the solution of nonconvex MINLPs have the potential to play an important role in the near future. Some very large problems can probably be solved only through heuristics, but what is more interesting is that Branch-and-Bound solvers for MINLPs are trying to reach a maturity level similar to those for MILPs, and one of the main differences between the two is the number of available primal heuristics. Obviously, obtaining good feasible solutions is critical for the performance of a Branch-and-Bound, and to this end commercial MILP codes have sometimes tens of fast heuristics. This trend has increased in recent years, and nonconvex MINLP solvers should probably follow the same steps if they want to be perceived as reliable and effective as their linear counterparts. Our VNS algorithm could be used within a Branch-and-Bound solver as a primal heuristic. There is of course room for improvement in the algorithm itself. It would be desirable to be able to find a first feasible solution as soon as possible; currently, we rely on a convex Branch-and-Bound solver used in a heuristic way to perform this task. The starting point provided to the solver determines the chances of finding such a solution. We believe that employing constraint programming techniques to round to the nearest integer as many fractional integer variables as possible, while still maintaining constraint feasibility, could greatly help. We also plan to test different solvers to evaluate performance as a stand-alone heuristic, and to reduce the number of parameters of the algorithm; preliminary tests show that a different combination of solvers through the main phase of the algorithm yields significantly better solution quality, although at the expense of a longer CPU time.

Acknowledgements

First of all, I want to thank Leo Liberti for giving me the possibility of working on this thesis: he always encouraged me to try harder, and much of this work would not have been done if it were not for him. I could not hope for a better advisor.

But he is not the only one who deserves credit. Philippe Baptiste, Philippe Goudal and Daniel Krob played an important part in advising me, although with different roles, and so did all the people I often worked with, especially: Benjamin Becquet, Gerard Cornuéjols, Daniel Delling, Nenad Mladenović, Dominik Schultes. They made this thesis possible. I also want to thank everyone at Mediamobile for always being helpful and friendly, and Tapio Westerlund for carefully checking through some misprints on the MINLPLib website.

Last but not least, the only people I could always count on, despite many difficulties: my family and my friends. I will not name each and every one of you, but you know who you are. Thanks for everything.

Disclaimer

Parts of this thesis have appeared, or are going to appear, in the following works: (38; 43; 44; 99; 109; 110; 111; 112; 113; 114).

References

[1] K. Aardal, R. E. Bixby, C. A. J. Hurkens, A. K. Lenstra, and J. W. Smeltink. Market split and basis reduction: Towards a solution of the Cornuéjols-Dawande instances. *INFORMS Journal on Computing*, 12(3):192–202, 2000.

[2] K. Abhishek, S. Leyffer, and J. Linderoth. Filmint: An outer-approximation based solver for nonlinear mixed-integer programs. Technical Report ANL/MCS-P1374-0906, Argonne National Laboratory, 2007.

[3] C. Adjiman, I. Androulakis, and C. Floudas. Global optimization of MINLP problems in process synthesis and design. *Computers & Chemical Engineering*, 21:S445–S450, 1997.

[4] R. Ahuja, T. Magnanti, and J. Orlin. *Network flows: theory, algorithms, and applications*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[5] R. Ahuja, J. Orlin, S. Pallottino, and M. Scutellà. Dynamic shortest paths minimizing travel times and costs. *Networks*, 41(4):197–205, 2003.

[6] M. Ajtai. The shortest vector problem in l_2 is NP-hard for randomized reductions. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing*, Dallas, TX, 1998.

[7] *Proceedings of the 8th Workshop on Algorithm Engineering and Experiments (ALENEX 06)*, Lecture Notes in Computer Science. Springer, 2006.

[8] K. Andersen, G. Cornuéjols, and Y. Li. Reduce-and-split cuts: Improving the performance of mixed integer Gomory cuts. *Management Science*, 51(11):1720–1732, 2005.

[9] M. Aouchiche, J. Bonnefoy, A. Fidahoussen, G. Caporossi, P. Hansen, L. Hiesse, J. Lacheré, and A. Monhait. VNS for extremal graphs 14: The AGX 2 system. In Liberti and Maculan (98), pages 281–308.

[10] E. Balas. Intersection cuts - a new type of cutting planes for integer programming. *Operations Research*, 19(1):19–39, 1971.

[11] E. Balas. Disjunctive programming. *Annals of Discrete Mathematics*, 5:3–51, 1979.

[12] E. Balas, S. Ceria, and G. Cornuéjols. Mixed 0-1 programming by lift-and-project in a branch-and-cut framework. *Management Science*, 42(9):1229–1246, 1996.

[13] E. Balas and A. Saxena. Optimizing over the split closure. *Mathematical Programming*, 113(2):219–240, 2008.

[14] C. Barrett, K. Bisset, M. Holzer, G. Konjevod, M. Marathe, and D. Wagner. Engineering label-constrained shortest-path algorithms. In *AAIM '08: Proceedings of the 4th international conference on Algorithmic Aspects in Information and Management*, pages 27–37, Berlin, Heidelberg, 2008. Springer-Verlag.

[15] C. Barrett, R. Jacob, and M. Marathe. Formal language constrained path problems. *SIAM Journal of computing*, 30(3):809–837, 2001.

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

[17] V. Batz, R. Geisberger, and P. Sanders. Time dependent contraction hierarchies — basic algorithmic ideas. Technical report, Universität Karlsruhe (TH), 2008. Available from World Wide Web: `http://arxiv.org/abs/0804.3947`.

[18] R. Bauer and D. Delling. SHARC: Fast and Robust Unidirectional Routing. In *Proceedings of the 10th Workshop on Algorithm Engineering and Experiments (ALENEX 08)*, pages 13–26. SIAM, 2008.

[19] R. Bauer, D. Delling, P. Sanders, D. Schieferdecker, D. Schultes, and D. Wagner. Combining hierarchical and goal-directed speed-up techniques for dijkstra's algorithm. In McGeoch (104), pages 303–318.

[20] M. S. Bazaraa and R. W. Langley. A dual shortest path algorithm. *SIAM Journal on Applied Mathematics*, 26(3):496–501, 1974.

[21] M. Beckmann, C. McGuire, and C. Winsten. Studies in the economics of transportation. Technical Report RM-1488, RAND Corporation, 1955.

[22] P. Belotti. Couenne, an open-source solver for mixed-integer nonconvex problems. In preparation.

[23] P. Belotti, J. Lee, L. Liberti, F. Margot, and A. Wächter. Branching and bounds tightening techniques for non-convex MINLP. Technical Report RC24620, IBM, 2008. Available from World Wide Web: `http://www.optimization-online.org/DB_HTML/2008/08/2059.html`.

[24] P. Bonami, L. Biegler, A. Conn, G. Cornuéjols, I. Grossmann, C. Laird, J. Lee, A. Lodi, F. Margot, N. Sawaya, and A. Wächter. An algorithmic framework for convex Mixed Integer Nonlinear Programs. Technical Report RC23771, IBM Corporation, 2005.

[25] P. Bonami, G. Cornuéjols, A. Lodi, and F. Margot. A feasibility pump for Mixed Integer Nonlinear Programs. Technical Report RC23862 (W0602-029), IBM Corporation, 2006.

[26] P. Bonami and J. Lee. `BONMIN` user's manual. Technical report, IBM Corporation, June 2007.

[27] J. Brimberg and N. Mladenović. A variable neighbourhood algorithm for solving the continuous location-allocation problem. *Studies in Location Analysis*, 10:1–12, 1996.

[28] A. Brook, D. Kendrick, and A. Meeraus. Gams, a user's guide. *ACM SIGNUM Newsletter*, 23(3-4):10–11, 1988.

[29] L. Buriol, M. Resende, and M. Thorup. Speeding up dynamic shortest path algorithms. *INFORMS Journal on Computing*, accepted for publication. Available from World Wide Web: `http://www.research.att.com/~mgcr/doc/dspa.pdf`.

[30] M. R. Bussieck, A. S. Drud, and A. Meeraus. MINLPLib — a collection of test models for Mixed-Integer Nonlinear Programming. *INFORMS Journal on Computing*, 15(1), 2003.

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

[32] I. Chabini and S. Lan. Adaptations of the A^* algorithm for the computation of fastest paths in deterministic discrete-time dynamic networks. *IEEE Transactions on Intelligent Transportation Systems*, 3(1):60–74, 2002.

[33] Coin-or branch-and-cut. Available from World Wide Web: `https://projects.coin-or.org/Cbc`.

[34] Coin-or cut generation library. Available from World Wide Web: `https://projects.coin-or.org/Cgl`.

[35] A. Consulting and Development. *SBB Release Notes*, 2002.

[36] K. Cooke and E. Halsey. The shortest route through a network with time-dependent internodal transit times. *Journal of Mathematical Analysis and Applications*, 14:493–498, 1966.

[37] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, second edition, 2001.

[38] G. Cornuéjols, L. Liberti, and G. Nannicini. Improved strategies for branching on general disjunctions. Technical Report 2071, Optimization Online, 2008. Available from World Wide Web: `http://www.optimization-online.com`.

[39] C. Daganzo. Reversibility of time-dependent shortest path problem. Technical report, Institute of Transportation Studies, University of California, Berkeley, 1998. Available from World Wide Web: `http://repositories.cdlib.org/its/reports/UCB-ITS-RR-98-14`.

[40] B. C. Dean. Continuous-time dynamic shortest path algorithms. Master's thesis, Massachussets Institute of Technology, 1999.

[41] D. Delling. Time-Dependent SHARC-Routing. In *Proceedings of the 16th Annual European Symposium on Algorithms (ESA'08)*, volume 5193 of *Lecture Notes in Computer Science*, pages 332–343. Springer, Sept. 2008.

[42] D. Delling. *Engineering and Augmenting Route Planning Algorithms*. PhD thesis, Fakultät für Informatik, Universität Fridericiana zu Karlsruhe (TH), Germany, Feb. 2009.

[43] D. Delling and G. Nannicini. Bidirectional Core-Based Routing in Dynamic Time-Dependent Road Networks. In S.-H. Hong, H. Nagamochi, and T. Fukunaga, editors, *Proceedings of the 19th International Symposium on Algorithms and Computation (ISAAC 08)*, volume 5369 of *Lecture Notes in Computer Science*, pages 813–824. Springer, 2008.

[44] D. Delling and G. Nannicini. Core routing on dynamic time-dependent road networks. Technical Report 2156, Optimization Online, 2008. Available from World Wide Web: `http://www.optimization-online.com`.

[45] D. Delling, P. Sanders, D. Schultes, and D. Wagner. Highway Hierarchies Star. In C. Demetrescu, A. V. Goldberg, and D. S. Johnson, editors, *Shortest Paths: Ninth DIMACS Implementation Challenge*, DIMACS Book. American Mathematical Society, 2008. accepted for publication, to appear.

[46] D. Delling and D. Wagner. Landmark-based routing in dynamic graphs. In Demetrescu (47), pages 52–65.

[47] C. Demetrescu, editor. *6th Workshop on Experimental Algorithms*, volume 4525 of *LNCS*, New York, 2007. Springer.

[48] R. B. Dial. Algorithm 360: shortest-path forest with topological ordering [h]. *Communications of the ACM*, 12(11):632–633, 1969.

[49] E. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[50] R. Dionne. Étude et extension d'un algorithme de Murchland. *INFOR*, 16:132–146, 1978.

[51] M. Dražic, V. Kovačević-Vujčić, M. Čangalović, and N. Mladenović. Glob — a new VNS-based software for global optimization. In Liberti and Maculan (98), pages 135–154.

[52] M. Drazić, C. Lavor, N. Maculan, and N. Mladenović. A continuous VNS heuristic for finding the tridimensional structure of a molecule, 2004.

[53] S. Dreyfus. An appraisal of some shortest-path algorithms. *Operations Research*, 17(3):395–412, 1969.

[54] M. Duran and I. Grossmann. An outer-approximation algorithm for a class of mixed-integer nonlinear programs. *Mathematical Programming*, 36:307–339, 1986.

[55] G. Fandel and T. Gal, editors. *Multiple Criteria Decision Making – Theory and Applications*, volume 177 of *Lecture Notes in Economics and Mathematical Systems*. Springer Verlag, Berlin, 1980.

[56] M. Fischetti and A. Lodi. Local branching. *Mathematical Programming*, 98:23–37, 2003.

[57] R. Fletcher and S. Leyffer. Solving Mixed Integer Nonlinear Programs by outer approximation. *Mathematical Programming*, 66:327–349, 1994.

[58] R. Fletcher and S. Leyffer. Numerical experience with lower bounds for MIQP branch-and-bound. *SIAM Journal of Optimization*, 8(2):604–616, 1998.

[59] R. Fletcher and S. Leyffer. User manual for filter. Technical report, University of Dundee, UK, Mar. 1999.

[60] L. R. Ford and D. R. Fulkerson. *Modern Heuristic Techniques for Combinatorial Problems*. Princeton University Press, Princeton, NJ, 1962.

[61] R. Fourer and D. Gay. *The AMPL Book*. Duxbury Press, Pacific Grove, 2002.

[62] M. Fredman and R. Tarjan. Fibonacci heaps and their use in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987.

[63] G. Gallo. Reoptimization procedures in shortest path problems. *Rivista di Matematica per le Scienze Economiche e Sociali*, 3:3–13, 1980.

[64] R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In McGeoch (104), pages 319–333.

[65] P. Gill. *User's guide for SNOPT version 7*. Systems Optimization Laboratory, Stanford University, California, 2006.

[66] P. Gill, W. Murray, and M. Saunders. Snopt: An sqp algorithms for large-scale constrained optimization. *SIAM Journal of Optimization*, 12(4):979–1006, 2002.

[67] A. Goldberg and C. Harrelson. Computing the shortest path: A^* meets graph theory. In *Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2005)*, pages 156–165, Philadelphia, 2005. SIAM.

[68] A. Goldberg, H. Kaplan, and R. Werneck. Reach for A^*: Efficient point-to-point shortest path algorithms. In goldberg (7), pages 129–143.

[69] A. Goldberg, H. Kaplan, and R. Werneck. Better landmarks within reach. In Demetrescu (47), pages 38–51.

[70] A. Goldberg and R. Werneck. Computing point-to-point shortest paths from external memory. In C. Demetrescu, R. Sedgewick, and R. Tamassia, editors, *Proceedings of the 7th Workshop on Algorithm Engineering and Experimentation (ALENEX 05)*, pages 26–40, Philadelphia, 2005. SIAM.

[71] A. V. Goldberg, H. Kaplan, and R. F. Werneck. Reach for A*: Shortest Path Algorithms with Preprocessing. In C. Demetrescu, A. V. Goldberg, and D. S. Johnson, editors, *Shortest Paths: Ninth DIMACS Implementation Challenge*, DIMACS Book. American Mathematical Society, 2008. accepted for publication, to appear.

[72] R. E. Gomory. An algorithm for integer solutions to linear programs. In P. Wolfe, editor, *Recent Advances in Mathematical Programming*, pages 269–302. McGraw-Hill, New York, 1963.

[73] Google maps API. Available from World Wide Web: `http://code.google.com/apis/maps/`.

[74] A. Halder. The method of competing links. *Transportation Science*, 4:36–51, 1970.

[75] A. Halder. Some new techniques in transportation planning. *Operational Research Quarterly*, 21:267–278, 1970.

[76] P. Hansen and N. Mladenović. Variable neighbourhood search: Principles and applications. *European Journal of Operations Research*, 130:449–467, 2001.

[77] P. Hansen and N. Mladenović. Variable neighbourhood search. In P. Pardalos and M. Resende, editors, *Handbook of Applied Optimization*. Oxford University Press, Oxford, 2002.

[78] P. Hansen and N. Mladenović. Variable neighbourhood search. In F. Glover and G. Kochenberger, editors, *Handbook of Metaheuristics*. Kluwer, Dordrecht, 2003.

[79] P. Hansen, N. Mladenović, and D. Urošević. Variable neighbourhood search and local branching. *Computers and Operations Research*, 33(10):3034–3045, 2006.

[80] E. Hart, N. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems, Science and Cybernetics*, SSC-4(2):100–107, 1968.

[81] M. Holzer, F. Schulz, and D. Wagner. Engineering multi-level overlay graphs for shortest-path queries. In wagneroverlay (7), pages 156–170.

[82] T. Ikeda, M. Tsu, H. Imai, S. Nishimura, H. Shimoura, T. Hashimoto, K. Tenmoku, and K. Mitoh. A fast algorithm for finding better routes by ai search techniques. In *Proceedings for the IEEE Vehicle Navigation and Information Systems Conference*, pages 291–296, 2004.

[83] ILOG. *ILOG CPLEX 11.0 User's Manual*. ILOG S.A., Gentilly, France, 2007.

[84] M. Karamanov and G. Cornuéjols. Branching on general disjunctions. Technical report, Carnegie Mellon University, 2005. Available from World Wide Web: `http://integer.tepper.cmu.edu`.

[85] D. E. Kaufman and R. L. Smith. Fastest paths in time-dependent networks for intelligent vehicle-highway systems application. *Journal of Intelligent Transportation Systems*, 1(1):1–11, 1993.

[86] B. S. Kerner. *The Physics of Traffic*. Springer, Berlin, 2004.

[87] J. Krarup and M. N. Rorbech. Lp formulations of the shortest path tree problem. *4OR*, 2:259–274, 2004.

[88] A. H. Land and A. G. Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28(3):497–520, 1960.

[89] C. Lavor, L. Liberti, and N. Maculan. Computational experience with the molecular distance geometry problem. In J. Pintér, editor, *Global Optimization: Scientific and Engineering Case Studies*. Springer, Berlin, 2006.

[90] S. Leyffer. User manual for MINLP_BB. Technical report, University of Dundee, UK, March 1999.

[91] L. Liberti. *Reformulation and Convex Relaxation Techniques for Global Optimization*. PhD thesis, Imperial College London, UK, Mar. 2004.

[92] L. Liberti. Writing global optimization software. In Liberti and Maculan (98), pages 211–262.

[93] L. Liberti. Reformulations in mathematical programming: Definitions and systematics. *RAIRO Operations Research*, to appear.

[94] L. Liberti, S. Cafieri, and F. Tarissan. Reformulations in mathematical programming: A computational approach. In A. Abraham, A.-E. Hassanien, and P. Siarry, editors, *Global Optimization: Theoretical Foundations and Applications*, Studies in Computational Intelligence. Springer, New York, to appear.

[95] L. Liberti and M. Dražic. Variable neighbourhood search for the global optimization of constrained NLPs. In *Proceedings of GO Workshop, Almeria, Spain*, 2005.

[96] L. Liberti, C. Lavor, and N. Maculan. Double VNS for the molecular distance geometry problem. In *Proc. of Mini Euro Conference on Variable Neighbourhood Search, Tenerife, Spain*, 2005.

[97] L. Liberti, C. Lavor, N. Maculan, and F. Marinelli. Double variable neighbourhood search with smoothing for the molecular distance geometry problem. *Journal of Global Optimization*, accepted for publication.

[98] L. Liberti and N. Maculan, editors. *Global Optimization: from Theory to Implementation*. Springer, Berlin, 2006.

[99] L. Liberti, G. Nannicini, and N. Mladenović. A good recipe for solving MINLPs. In V. Maniezzo, T. Stuetze, and S. Voss, editors, *MATHEURISTICS: Hybridizing metaheuristics and mathematical programming*, Operations Research/Computer Science Interface Series. Springer, 2008.

[100] L. Liberti, P. Tsiakis, B. Keeping, and C. Pantelides. *ooOPS*. Centre for Process Systems Engineering, Chemical Engineering Department, Imperial College, London, UK, 2001.

[101] A. Lodi. Personal communication, 2007.

[102] P. Loubal. A network evaluation procedure. *Highway Research Record*, 205:96–109, 1967.

[103] J. Maue, P. Sanders, and D. Matijevic. Goal directed shortest path queries using precomputed cluster distances. In C. Alvarez and M. J. Serna, editors, *WEA 2006*, volume 4007 of *LNCS*, pages 316–327, New York, 2006. Springer.

[104] C. McGeoch, editor. *Proceedings of the 8th Workshop on Experimental Algorithms (WEA 2008)*, volume 5038 of *Lecture Notes in Computer Science*, New York, 2008. Springer.

[105] N. Mladenović, J. Petrović, V. Kovačević-Vujčić, and M. Čangalović. Solving a spread-spectrum radar polyphase code design problem by tabu search and variable neighbourhood search. *European Journal of Operations Research*, 151:389–399, 2003.

[106] R. H. Möhring, H. Schilling, B. Schütz, D. Wagner, and T. Willhalm. Partitioning graphs to speed up dijkstra's algorithm. In S. E. Nikoletseas, editor, *WEA*, volume 3503 of *Lecture Notes in Computer Science*, pages 189–202. Springer, 2005.

[107] J. D. Murchland. The effect of increasing or decreasing the length of a single arc on all shortest distances in a graph. *London Businness School, Transport Network Theory Unit*, 1967.

[108] J. D. Murchland. *A fixed matrix method for all shortest distances in a directed graph and for the inverse problem*. PhD thesis, University of Karlsruhe, 1970.

[109] G. Nannicini, P. Baptiste, G. Barbier, D. Krob, and L. Liberti. Fast paths in large-scale dynamic road networks. *Computational Optimization and Applications*, 2008.

[110] G. Nannicini, P. Baptiste, D. Krob, and L. Liberti. Fast computation of point-to-point paths on time-dependent road networks. In B. Yang, D.-Z. Du, and C. Wang, editors, *Proceedings of the 2nd International Conference on Combinatorial Optimization and Applications (COCOA 08)*, volume 5165 of *LNCS*, pages 225–234, Berlin, 2008. Springer.

[111] G. Nannicini, P. Baptiste, D. Krob, and L. Liberti. Fast paths in dynamic road networks. In A. Quillot and P. Mahey, editors, *Proceedings of ROADEF 08*, Clermont-Ferrand, 2008. Université Blaise Pascal.

[112] G. Nannicini, D. Delling, L. Liberti, and D. Schultes. Bidirectional A^* search for time-dependent fast paths. In McGeoch (104), pages 334–346.

[113] G. Nannicini, D. Delling, L. Liberti, and D. Schultes. Bidirectional A^* search on time-dependent road networks. Technical Report 2154, Optimization Online, 2008. Available from World Wide Web: `http://www.optimization-online.com`.

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

[115] G. Nemhauser. A generalized permanent label setting algorithm for the shortest path between specified nodes. *Journal of Mathematical Analysis and Applications*, 38:328–334, 1972.

[116] T. NV. *Tele Atlas Multinet ShapeFile 4.3.1 Format Specifications*. TeleAtlas NV, May 2005.

[117] A. Orda and R. Rom. Shortest-path and minimum delay algorithms in networks with time-dependent edge-length. *Journal of the ACM*, 37(3):607–625, 1990. Available from World Wide Web: `citeseer.ist.psu.edu/orda90shortestpath.html`.

[118] J. Owen and S. Mehrotra. Experimental results on using general disjunctions in branch-and-bound for general-integer linear program. *Computational Optimization and Applications*, 20:159–170, 2001.

[119] S. Pallottino and M. Scutellà. Dual algorithms for the shortest path tree problem. *Networks*, 29:125–133, 1997.

[120] S. Pallottino and M. Scutellà. A new algorithm for reoptimizing shortest paths when the arc costs change. *Operations Research Letters*, 31(2):149–160, 2003.

[121] F. Pellegrini. SCOTCH 5.0 user guide. Technical report, Laboratoire Bordelais de Recherche en Informatique, 2007. Available from World Wide Web: `http://www.labri.fr/perso/pelegrin/scotch/`.

[122] J. Puchinger and G. Raidl. Relaxation guided variable neighbourhood search. In *Proc. of Mini Euro Conference on Variable Neighbourhood Search, Tenerife, Spain*, 2005.

[123] E. Pyrga, F. Schulz, D. Wagner, and C. Zaroliagis. Efficient Models for Timetable Information in Public Transportation Systems. *ACM Journal of Experimental Algorithmics*, 12:Article 2.4, 2007.

[124] P. Sanders and D. Schultes. Highway hierarchies hasten exact shortest path queries. In G. Stølting Brodal and S. Leonardi, editors, *13th Annual European Symposium on Algorithms (ESA 2005)*, volume 3669 of *Lecture Notes in Computer Science*, pages 568–579. Springer, 2005.

[125] P. Sanders and D. Schultes. Engineering highway hierarchies. In *ESA 2006*, volume 4168 of *Lecture Notes in Computer Science*, pages 804–816. Springer, 2006.

[126] P. Sanders and D. Schultes. Dynamic highway-node routing. In Demetrescu (47), pages 66–79.

[127] P. Sanders and D. Schultes. Engineering fast route planning algorithms. In Demetrescu (47), pages 23–36.

[128] A. Schrijver. *Combinatorial Optimization: Polyhedra and Efficiency*. Springer, Berlin, 2003.

[129] D. Schultes. Fast and exact shortest path queries using highway hierarchies. *Master Thesis, Informatik, Universität des Saarlandes*, June 2005.

[130] R. Sedgewick and J. Vitter. Shortest paths in euclidean graphs. *Algorithmica*, 1(1):31–48, 1986.

[131] E. Smith and C. Pantelides. A symbolic reformulation/spatial branch-and-bound algorithm for the global optimisation of nonconvex MINLPs. *Computers & Chemical Engineering*, 23:457–478, 1999.

[132] M. Tawarmalani and N. Sahinidis. Global optimization of mixed integer nonlinear programs: A theoretical and computational study. *Mathematical Programming*, 99:563–591, 2004.

[133] D. Wagner and T. Willhalm. Speed-up techniques for shortest-path computations. In W. Thomas and P. Weil, editors, *24th Annual Symposium on Theoretical Aspects of Computer Science (STACS 2007)*, volume 4393 of *LNCS*, pages 23–36, New York, 2007. Springer.

[134] D. Wagner, T. Willhalm, and C. Zaroliagis. Geometric containers for efficient shortest-path computation. *ACM Journal of Experimental Algorithmics*, 10:1–30, 2005.

[135] T. Westerlund. Some transformation techniques in global optimization. In Liberti and Maculan (98), pages 45–74.

[136] T. Westerlund and R. Pörn. Solving pseudo-convex mixed integer optimization problems by cutting plane techniques. *Optimization and Engineering*, 3:235–280, 2002.

[137] T. Westerlund, H. Skrifvars, I. Harjunkoski, and R. Pörn. An extended cutting plane method for a class of non-convex MINLP problems. *Computers & Chemical Engineering*, 22(3):357–365, 1998.