# 13th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems

**ATMOS'13, September 5, 2013, Sophia Antipolis, France**

Edited by

Daniele Frigioni
Sebastian Stiller

OASICS

*Editors*

| | |
|---|---|
| Daniele Frigioni | Sebastian Stiller |
| University of L'Aquila | Technische Universität Berlin |
| L'Aquila, Italy | Berlin, Germany |
| `daniele.frigioni@univaq.it` | `sebastian.stiller@tu-berlin.de` |

*ACM Classification 1998*
F.2 Analysis of Algorithms and Problem Complexity, G.1.6 Optimization, G.2.2 Graph Theory, G.2.3 Applications

**ISBN 978-3-939897-58-3**

*Published online and open access by*
Schloss Dagstuhl –Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at http://www.dagstuhl.de/dagpub/978-3-939897-58-3.

*Publication date*
September, 2013

*Bibliographic information published by the Deutsche Nationalbibliothek*
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at http://dnb.d-nb.de.

Digital Object Identifier: 10.4230/OASIcs.ATMOS.2013.i

**ISSN 2190-6807**

**www.dagstuhl.de/oasics**

# Contents

13th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS'13).
Editors: Daniele Frigioni, Sebastian Stiller
OASICS OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

# Preface

Transportation networks give rise to very complex and large-scale network optimization problems requiring innovative solution techniques and ideas from mathematical optimization, theoretical computer science, and operations research. Since 2000, the series of *Algorithmic Approaches for Transportation Modelling, Optimization, and Systems* (ATMOS) workshops brings together researchers and practitioners who are interested in all aspects of algorithmic methods and models for transportation optimization and provides a forum for the exchange and dissemination of new ideas and techniques. The scope of ATMOS comprises all modes of transportation.

The 13th ATMOS workshop (ATMOS'13) was held in connection with ALGO'13, by INRIA and Campus SophiaTech, in Sophia Antipolis, France, on September 5, 2013. Topics of interest for ATMOS'13 were all optimization problems for passenger and freight transport, including, but not limited to, Demand Forecasting, Models for User Behavior, Design of Pricing Systems, Infrastructure Planning, Multi-modal Transport Optimization, Mobile Applications for Transport, Congestion Modeling and Reduction, Line Planning, Timetable Generation, Routing and Platform Assignment, Vehicle Scheduling, Route Planning, Crew and Duty Scheduling, Rostering, Delay Management, Routing in Road Networks, Traffic Guidance. Of particular interest were papers applying and advancing the following techniques: graph and network algorithms, combinatorial optimization, mathematical programming, approximation algorithms, methods for the integration of planning stages, stochastic and robust optimization, online and real-time algorithms, algorithmic game theory, heuristics for real-world instances, simulation tools.

In response to the call for papers we received 26 submissions, all of which were reviewed by at least three referees. The submissions were judged on originality, technical quality, and relevance to the topics of the workshop. Based on the reviews, the program committee selected the 12 papers which appear in this volume. Together, they quite impressively demonstrate the range of applicability of algorithmic optimization to transportation problems in a wide sense. In addition, Tobias Harks kindly agreed to complement the program with an invited talk entitled *Modeling and Optimizing Traffic Networks*.

We would like to thank the members of the Steering Committee of ATMOS for giving us the opportunity to serve as Program Chairs of ATMOS'13, all the authors who submitted papers, Tobias Harks for accepting our invitation to present an invited talk, the members of the Program Committee and all the additional reviewers for their valuable work in selecting the papers appearing in this volume, and the local organizers for hosting the workshop as part of ALGO'13. We also acknowledge the use of the EasyChair system for the great help in managing the submission and review processes, and Schloss Dagstuhl for publishing the proceedings of ATMOS'13 in its OASIcs series.

Finally, we are pleased to announce that this year, for the first time, ATMOS PC awards a Best Paper Award. The Best Paper of ATMOS 2013 is *A Configuration Model for the Line Planning Problem* by Ralf Borndörfer, Heide Hoppmann, and Marika Karbstein.

September, 2013

Daniele Frigioni

Sebastian Stiller

# Organization

## Program Committee

| | |
|---|---|
| Ralf Borndörfer | Zuse-Institut and FU Berlin, Germany |
| Daniel Delling | Microsoft Research Silicon Valley, USA |
| Daniele Frigioni (co-chair) | University of L'Aquila, Italy |
| Laura Galli | University of Pisa, Italy |
| Spyros Kontogiannis | University of Ioannina, Greece |
| Christian Liebchen | Deutsche Bahn, Germany |
| Gabor Maroti | VU Amsterdam and Netherlands Railways, The Netherlands |
| Frédéric Meunier | Ecole des Ponts ParisTech, France |
| Dario Pacciarelli | Roma Tre University, Italy |
| Marc Pfetsch | TU Darmstadt, Germany |
| Robert Shorten | IBM Research and The Hamilton Institute, Ireland |
| Sebastian Stiller (co-chair) | TU Berlin, Germany |

## Steering Committee

| | |
|---|---|
| Alberto Marchetti-Spaccamela | Sapienza University of Rome, Italy |
| Rolf Möhring | TU Berlin, Germany |
| Dorothea Wagner | Karlsruhe Institute of Technology, Germany |
| Christos Zaroliagis | University of Patras, Greece |

## List of Additional Reviewers

Gianlorenzo D'Angelo, Andrea D'Ariano, Mattia D'Emidio, Julian Dibbelt, Pavlos Efraimidis, Dimitris Fotakis, Loukas Georgiadis, Jan Marecek, Martin Mevissen, Alfredo Navarra, Alexander Richter, Arieh Schlote, Karsten Weihe, Jia Wuan Yu, Christos Zaroliagis.

## Local Organizing Committee

Frédéric Cazals, Agnès Cortell (event manager), David Coudert, Olivier Devillers, Joanna Moulierac, Monique Teillaud (chair).

# List of Authors

Christian Artigues

Hannah Bast

Arthur Bit-Monnot

Katerina Böhmová

Ralf Borndörfer

Mirko Brodesser

Emilio Carrizosa

Donatella Firmani

Daniele Frigioni

Marc Goerigk

Jonas Harbering

Sascha Heße

Heide Hoppmann

Marie-José Huguet

Giuseppe F. Italiano

Nicolas Jozefowiez

Marika Karbstein

Marc-Olivier Killijian

Moritz Kobitzsch

Luigi Laura

Alexander Lazarev

Matus Mihalák

Matthias Müller-Hannemann

Andreas Paraskevopoulos

Tobias Pröger

Marcel Radermacher

Ruslan Sadykov

Federico Santaroni

Boadu Mensah Sarpong

Dennis Schieferdecker

Marie Schmidt

Anita Schöbel

Vitaliy Shiryaev

Rastislav Šrámek

Jonas Sternisko

Sebastian Stiller

Sabine Storandt

Alexey Stratonnikov

Peter Widmayer

Christos Zaroliagis

13th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS'13).
Editors: Daniele Frigioni, Sebastian Stiller
OASICS OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

# Recoverable Robust Timetable Information *

**Marc Goerigk[†1], Sascha Heße[2], Matthias Müller-Hannemann[2], Marie Schmidt[3], and Anita Schöbel[3]**

**1 Fachbreich Mathematik**
**Technische Universität Kaiserslautern, Germany**
goerigk@mathematik.uni-kl.de

**2 Institut für Informatik**
**Martin-Luther-Universität Halle-Wittenberg, Germany**
sascha.hesse@student.uni-halle.de, muellerh@informatik.uni-halle.de

**3 Institut für Numerische und Angewandte Mathematik**
**Georg-August Universität Göttingen, Germany**
{m.schmidt,schoebel}@math.uni-goettingen.de

## Abstract

Timetable information is the process of determining a suitable travel route for a passenger. Due to delays in the original timetable, in practice it often happens that the travel route cannot be used as originally planned. For a passenger being already en route, it would hence be useful to know about alternatives that ensure that his/her destination can be reached.

In this work we propose a *recoverable robust* approach to timetable information; i.e., we aim at finding travel routes that can easily be updated when delays occur during the journey. We present polynomial-time algorithms for this problem and evaluate the performance of the routes obtained this way on schedule data of the German train network of 2013 and simulated delay scenarios.

**1998 ACM Subject Classification** F.2.2 Nonnumerical Algorithms and Problems, G.2.2 Graph Theory (Graph algorithms, Network problems)

**Keywords and phrases** timetable information, recoverable robustness, delay scenarios

**Digital Object Identifier** 10.4230/OASIcs.ATMOS.2013.1

## 1 Introduction

In timetable information, the following problem is typically considered: Given a timetable, an origin and destination, and an earliest departure time, find the "best" route leading from origin to destination; see [21] for a survey. An obvious criterion to evaluate the quality of a route is its duration (or travel time); however, many other criteria have been suggested, as, e.g., the number of changes or the ticket costs [22, 14, 4]. Also the reliability of a path has been considered as a means to account for delays [14, 20, 22]. In [13], decision trees for passengers' travels under uncertainty are constructed. In a recent work [17, 18], approaches from the field of robust optimization were considered. Robust optimization is an approach to handle uncertainty in optimization problems that dates back to the 70s [24].

* partially supported by grants SCHO 1140/3-1 and MU 1482/4-3 within the DFG programme SPP 1307 *Algorithm Engineering* and by the European Union Seventh Framework Programme (FP7-PEOPLE-2009-IRSES) under grant number 246647 and the New Zealand Government (project OptALI).

† Effort sponsored by the Air Force Office of Scientific Research, Air Force Material Command, USAF, under grant number FA8655-13-1-3066. The U.S Government is authorized to reproduce and distribute reprints for Governmental purpose notwithstanding any copyright notation thereon.

During the late 90s, it received new attention through the work of Ben-Tal, Nemirovski and co-authors [2, 3], that sparked a manifold of concepts and algorithms; among them the $\Gamma$-approach of [5], adjustable robustness [1], light robustness [16], or recoverable robustness [19, 25]. In our work we focus on recoverable robustness. This is a two-stage concept: Given a set of recovery algorithms, a solution is considered as being robust when for every scenario it can be "repaired" using an recovery algorithm to become feasible. An application to the uncertain shortest path problem has been considered in [6], where the set of recovery algorithms is given by exchanging up to a constant $K$ arcs of the path. Related work can be found in [23], where a given path is updated to a new solution by either using or removing $k$ arcs. Further applications of recoverable robustness include shunting [10], timetabling [11, 25, 19], platforming [9, 25, 19], the empty repositioning problem [15], railway rolling stock planning [8] and the knapsack problem [7]. In some previous work [17, 18], robust passenger information has been considered. It was shown that finding a strictly robust travel route which hedges against any possible delay scenario is an NP-hard problem and for practical application much too conservative. As an alternative, a robustness concept based on light robustness has been proposed. However, it is assumed that a passengers stays on the planned route whatever happens. In contrast to this, we allow that a passengers changes his/her route even if he/she already started the journey.

**Contributions.** In timetable information, as in many other problems, the passenger does not know the scenarios from the beginning of his/her trip, but learns the current scenario en route. This aspect has been neglected in previous work. In this paper, we describe a recoverable robustness approach to the timetable information problem which takes into account that the actual scenario is learned at some time point en route, and that the travel route may be updated from this point on. For such a recovery, all possible alternative routes may be chosen. The goal is to include this recovery step in the planning phase, i.e. to find a travel route which may be recovered for every delay scenario from a given uncertainty set.

Furthermore, our approach can deal with complicated delay scenarios, as they occur in public transportation where source delays cause the dropping of transfers and changes in the durations of driving and waiting activities. We develop polynomial-time algorithms that can handle any finite set of scenarios and test them on delay scenarios that are generated by propagating delay in transportation systems.

Using large-scale data modeling the train network of Germany, we show the effectiveness of our approach.

**Overview.** The remainder of this work is structured as follows: We shortly recapture the nominal timetable information problem, and introduce our recoverable robust model in Section 2. We present a polynomial-time label-setting algorithm in Section 3, and demonstrate its applicability to German railway data provided by Deutsche Bahn AG in Section 4. We conclude the paper and discuss further research directions in Section 5.

## 2 Model and Notation

### 2.1 Timetable information

In the following we refer to *train timetables* for the sake of simplicity; however, all results can be transferred to any other type of public transport. The starting point for our considerations is a directed acyclic graph, the so-called *event-activity network* (EAN) $\mathcal{N} = (\mathcal{E}, \mathcal{A})$ which is regarded over a finite time horizon. Nodes $\mathcal{E}$ represent events in the train schedule: They can either be

- arrival events $\mathcal{E}_{arr}$ (modeling the arrival of a certain train at a certain station), or
- departure events $\mathcal{E}_{dep}$ (modeling the departure of a certain train from a certain station).

Events are connected by directed arcs, the *activities*, which can be either

- driving activities $\mathcal{A}_{drive}$ (modeling the trip of a train from a departure event to an arrival event),
- waiting activities $\mathcal{A}_{wait}$ (modeling the time a train spends between an arrival and a departure event for passengers to embark and disembark),
- or transfer activities $\mathcal{A}_{trans}$ (modeling passenger movements from one arrival event to another departure event within the same station).

Each event $i \in \mathcal{E}$ has a schedule time $\pi_i \in \mathbb{N}$; furthermore, to compute how delays spread within this network (see Section 4.2), we may assume that for each activity $(i, j) \in \mathcal{A}$ a minimal duration $l_{ij}$ is known, and thus a buffer time $b_{ij} := \pi_j - \pi_i - l_{ij}$. We assume that the initial timetable $\pi$ is feasible, i.e., $\pi_j - \pi_i \geq l_{ij}$, hence all buffer times are nonnegative. The timetable information problem consists of finding a path within the event-activity network from one station to another, given an earliest departure time $s$. More precisely, we introduce two virtual events, namely one origin event $u$ and one destination event $v$, corresponding to a given origin station $s_u$ and a destination station $s_v$. The origin event $u$ is connected by origin activities $\mathcal{A}_{org}$ with all departure events at station $s_u$ taking place not earlier than $s$, while all arrival events at station $s_v$ are connected with $v$ by destination activities $\mathcal{A}_{dest}$. We need to find a path $P$ from $u$ to $v$ in $\mathcal{N}$ such that the nominal travel time $t_{nom}(P) := \pi_{\text{last}(P)} - s$ on $P$ is minimal, where last$(P)$ denotes the last arrival event on $P$.

## 2.2 Delays

Paths with minimal travel time in the EAN may be vulnerable to delays, i.e., in case of delays, the originally planned path may take much longer than planned, or planned transfers may even become infeasible if the connecting train does not wait for a delayed feeder train.

The aim of this paper is to give robust timetable information, i.e., to find paths in the EAN which are less vulnerable to delays. The delays observed in a public transportation system originate from source delays $d_a$ which can occur on the driving and waiting activities $a$ of the train. These delays are partially absorbed by buffer times on the activities, however, they *propagate* through the network to subsequent events along driving and waiting activities and – if a transfer is maintained – along the corresponding transfer activity. We assume that each transfer is assigned a *waiting time* which specifies how long the connecting train will wait for the feeder train. If the delay of the feeder train exceeds the waiting time, the connecting train will depart on time. See Section 4.2 for details on our *delay propagation* method. We denote by $\mathcal{A}_{\text{transfer}}(d)$ the set of maintained transfer activities in scenario $d$ and denote the *delay network* $\mathcal{N}(d) := (\mathcal{E}, \mathcal{A}(d))$ with $\mathcal{A}(d) := \mathcal{A}_{\text{drive}} \cup \mathcal{A}_{\text{wait}} \cup \mathcal{A}_{\text{transfer}}(d)$. The updated timetable is denoted by $\pi(d)$. In this paper, we make the (simplifying) assumption that at some point in time, the passenger learns about *all* delays and can adapt ('recover') his/her travel route accordingly. We partition the events of the networks in a set $U^\xi$ of events where no delay has occurred so far and the passenger has not learned about future delays and a set $V^\xi$ where he/she knows all delays. We require the following properties of an *information scenario* $\xi = (\mathcal{N}^\xi, \pi^\xi, U^\xi, V^\xi)$ consisting of a delay network $\mathcal{N}^\xi$, a disposition timetable $\pi^\xi$ on this network, and a partition $(U^\xi, V^\xi)$ of the events $\mathcal{E}$:

- $u \in U^\xi$, $v \in V^\xi$,
- if $\pi^\xi_j > \pi_j$, $j$ is in $V^\xi$,
- all $i$ with $(i, v) \in \mathcal{A}_{dest}$ are contained in $V^\xi$,
- if $i$ is in $V^\xi$, all successors of $i$ are in $V^\xi$.

A way to define the partition $(U^\xi, V^\xi)$ between nodes $U^\xi$ where no delay information is available and nodes $V^\xi$ with full delay information is to set $U^\xi := \{j : \pi_j < t^\xi\}$, $V^\xi := \{j : \pi_j \geq t^\xi\}$, where $t^\xi$ denotes a *revealing time* $t^\xi \leq \min_{j \in \mathcal{E}: \pi_j^\xi - \pi_j > 0} \pi_j$ for every scenario $\xi$. For our computational experiments, we obtain $\mathcal{N}(\xi) := \mathcal{N}(d^\xi)$ and $\pi^\xi := \pi(d^\xi)$ by delay propagation, see Section 4.2. However, our methods work for any set of scenarios $\xi = (\mathcal{N}^\xi, \pi^\xi, U^\xi, V^\xi)$ as described above; it is not necessary to know the source delays to apply them. We define the set of activities where scenario $\xi$ is revealed as $\mathcal{A}^\xi := \{(i,j) \in \mathcal{A} : i \in U^\xi, j \in V^\xi\}$. A set of information scenarios will be called an *uncertainty set* and denoted by $\mathcal{U}$. In this paper, we consider only finite uncertainty sets.

## 2.3 Recoverable Robust Timetable Information

Intuitively, we will call a path $P$ recoverable robust if, when an information scenario $\xi$ occurs while a passenger is traveling on $P$, this passenger can take a *recovery path* $P^\xi$, to his/her destination. To formally define recoverable robust paths, we make use of the following observation: Let $\mathcal{U}$ be an uncertainty set and let $P$ be a path from $u$ to $v$ in $\mathcal{N}$.

▶ **Lemma 1.** *For every $\xi \in \mathcal{U}$, $P$ contains exactly one arc from $\mathcal{A}^\xi$.*

We denote this arc by $(i^\xi(P), j^\xi(P))$. We denote by $\mathcal{Q}^\xi(j)$ the set of *recovery paths*, i.e., all paths from a node $j$ to $v$ in $\mathcal{N}^\xi$, and set $\mathcal{Q}^\xi(P) := \mathcal{Q}^\xi(j^\xi(P))$.

▶ **Definition 2.** A path $P$ is called *recoverable robust* (with respect to uncertainty set $\mathcal{U}$) if for any $\xi \in \mathcal{U}$ the set of recovery paths $\mathcal{Q}^\xi(P)$ is not empty.

We assume that the passenger travels on the chosen path $P$ until he/she learns about the information scenario he/she is in, i.e., until node $j^\xi(P)$. Since at this node, the full information of $\xi$, i.e., $\mathcal{N}^\xi$, $\pi^\xi$, $U^\xi$ and $V^\xi$ is revealed to the passenger, he/she can take the best path for this scenario. Thus, we assume that he/she reroutes from his/her current position according to scenario $\xi$.

The goal of this paper is to find "good" recoverable robust paths. However, there are different ideas on how to measure the quality of a recoverable robust path. We can evaluate

- the *nominal quality*: which recoverable robust path has shortest travel time if no delays occur?
- the *worst-case quality*: which recoverable robust path has the earliest guaranteed arrival time?

Hence, we consider the following bicriteria problem:

▶ **Problem 1.** *Bicriteria recoverable robust paths*
**Input:** EAN $\mathcal{N} = (\mathcal{E}, \mathcal{A})$ with timetable $\pi$, origin $u$ and destination $v$, starting time $s$, and uncertainty set $\mathcal{U}$.
**Task:** Find a path $P$ from $u$ to $v$ in $\mathcal{N}$ which is recoverable robust and minimizes

1. the nominal travel time $t_{nom}(P) = \pi_{\text{last}(P)} - s$ where last$(P)$ is the last arrival node on $P$ *(the nominal objective function)*
2. the worst-case travel time $t_{wc}(P) = \max_{\xi \in \mathcal{U}} \min_{Q \in \mathcal{Q}^\xi(P)} \pi^\xi_{\text{last}(Q)} - s$ where last$(Q)$ is the last arrival event on $Q$ *(the worst-case objective function).*

Note that for simplicity, we call $t_{wc}(P)$ the worst-case travel time of $P$, although the path $P$ is only taken in the nominal case and an alternative path $P^\xi := \text{argmin}_{Q \in \mathcal{Q}^\xi(P)} \pi^\xi_{\text{last}(Q)} - s$ is taken in case of delay scenario $\xi$.

In other words, the bicriteria recoverable robust shortest path problem aims at finding paths which, on the one hand, are good in the nominal case, i. e., if no delays occur, and on the other hand hedge against the scenarios from the uncertainty set $\mathcal{U}$ by minimizing worst-case travel time on the corresponding recovery paths.

## 3 Algorithms for Recoverable Robust Paths

### 3.1 A Recovery-Label Setting Algorithm

In this section we show that in case of finite uncertainty sets solutions to the bicriteria recoverable robust path problem can be found as solutions to a bicriteria minimax bottleneck shortest path problem in the EAN with *recovery labels* $L(a) := (L_{nom}(a), L_{wc}(a))^T$ at all arcs $a \in \bigcup_{\xi\in\mathcal{U}} A^\xi$. The *minimax bottleneck shortest path problem* is the problem of finding a path between two nodes in a network which minimizes $\max_{a\in P} c(a)$ in a graph with edge labels $c(a)$. In the bicriteria version of this problem, every arc $a$ is assigned two different labels $c_1(a)$ and $c_2(a)$. We now state the preprocessing Algorithm 1 which calculates the recovery labels $L(a) := (L_{nom}(a), L_{wc}(a))^T$ needed to apply solution methods for the bicriteria minimax bottleneck shortest path problem. In Algorithm 1, for every $a = (i, j) \in \mathcal{A}^\xi$, $L^\xi(a)$ denotes the minimal travel time on a path which uses node $j$ in scenario $\xi$. If no such path exists in scenario $\xi$, $L^\xi(a)$ is set to $\infty$. The algorithm returns the labels $L = (L_{nom}, L_{wc})^T$ which are 0 for all $a \notin \bigcup_{\xi\in\mathcal{U}} A^\xi$. For $a \in \bigcup_{\xi\in\mathcal{U}} A^\xi$, $L_{nom}(a)$ denotes the minimum nominal travel time when using a path containing node $j$, (and is $\infty$, if no such path exists) while $L_{wc}(a)$ represents the worst-case travel time for scenarios revealed at node $j$.

After initialization of all required labels to the value 0 (lines 1-6), we compute the shortest path distance from every event to the destination in the nominal scenario (line 7). This can be done by a single invocation of a standard shortest path tree computation in the reversed digraph from the destination $v$. Then, in the for-loop of lines 8-15, we iterate over all delay scenarios. With respect to the revealing time of scenario $\xi$, we now determine the set $V^\xi$. Using again a backward shortest path tree computation with respect to $\mathcal{N}^\xi$, we determine for every event $j \in \mathcal{E}$ the length of a shortest path towards the destination $v$. Using these values, we can set the nominal and worst-case labels for paths which go through arcs in $\mathcal{A}^\xi$ (lines 11-13). For ease of notation, we use $\infty + k = \infty$ for all values $k$. Note that the label $L_{nom}(a)$ is only set if the corresponding edge $a$ can be used in some scenario $\xi \in \mathcal{U}$. We finally obtain the worst-case labels by taking the maximum over all scenarios. Note that lines 16-19 could be easily integrated into the main loop, but in the way presented here, the main loop can be run in parallel.

Given the recovery labels, the worst-case minimal travel time $t_{wc}(P)$ on a path $P$ can be calculated as the maximum over the labels $L_{wc}$ on $P$, as stated in the following lemma.

▶ **Lemma 3.** *Let $P$ be a path from $u$ to $v$ in $\mathcal{N}$. Then for the labels calculated in Algorithm 1 it holds that*

- *if $\max_{a\in P} L_{wc}(a) < \infty$, $P$ is recoverable robust, and*
- *$t_{wc}(P) = \max_{a\in P} L_{wc}(a)$.*

**Proof.** Consider an arbitrary scenario $\xi := (\mathcal{N}^\xi, \pi^\xi, U^\xi, V^\xi)$. The passenger travels on path $P$ until node $j^\xi(P)$. Then, he/she can take the path calculated in step 10 of the algorithm until node $v$ with total length $L^\xi(i^\xi(P), j^\xi(P))$ and this path has minimal length in $\mathcal{N}^\xi$ among all paths containing node $j$. We conclude that (1) $P$ is recoverable robust, and (2) $t_{wc}(P) = \max_{a\in P} L_{wc}(a)$. ◀

For any path $P$, the labels $L_{nom}$ constitute lower bounds on the nominal travel time on $P$. However, for an arbitrary path $P$, the nominal traveling time can exceed $\max_{a\in P} L_{nom}(a)$. This can be avoided for paths which do not make detours after the scenarios are revealed.

**Algorithm 1** Construction of recovery labels

```
Require: EAN $\mathcal{N} = (\mathcal{E}, \mathcal{A})$ with timetable $\pi$, origin node $u$, destination node $v$, starting
    time $s$, and finite uncertainty set $\mathcal{U}$.
Ensure: Label $L(a) \in \mathbb{R}^2_+$ for every $a \in \mathcal{A}$.
 1: for $(i,j) \in \mathcal{A}$ do                                          ▷ Initialization
 2:     Set $L_{nom}(i,j) := 0$.
 3:     for $\xi \in \mathcal{U}$ do
 4:         Set $L^\xi(i,j) := 0$.
 5:     end for
 6: end for
 7: Find length $K_{nom}(j)$ of shortest path from every $j \in \mathcal{E}$ to $v$ in $\mathcal{N}$. Set $K_{nom}(j) := \infty$ if
    no such path exists.
 8: for $\xi \in \mathcal{U}$ do
 9:     Determine $\mathcal{A}^\xi$.
10:     Find length $K^\xi_{wc}(j)$ of shortest path from every $j \in \mathcal{E}$ to $v$ in $\mathcal{N}^\xi$. Set $K^\xi_{wc}(j) := \infty$
        if no such path exists.
11:     for $(i,j) \in \mathcal{A}^\xi$ do
12:         Set $L_{nom}(i,j) := \pi_j - s + K_{nom}(j)$.                  ▷ Setting nominal labels.
13:         Set $L^\xi(i,j) := \pi^\xi_j - s + K^\xi_{wc}(j)$.               ▷ Setting worst-case labels.
14:     end for
15: end for
16: for $(i,j) \in \mathcal{A}$ do
17:     Set $L_{wc}(i,j) := \max_{\xi \in \mathcal{U}} L^\xi(i,j)$
18:     Set $L(i,j) := (L_{nom}(i,j), L_{wc}(i,j))^T$.
19: end for
20: return $L$
```

▶ **Lemma 4.** *Let $P$ be a path from $u$ to $v$ in $\mathcal{N}$ such that the path $P^2$ defined as the subpath of path $P$ starting in the last arc $(i,j)$ in $P \cap \left(\bigcup_{\xi \in \mathcal{U}} \mathcal{A}^\xi\right)$ is a shortest path from $j$ to $v$. Then for the labels calculated in Algorithm 1 it holds that*

- *if $\max_{a \in P} L_{wc}(a) < \infty$, $P$ is recoverable robust,*
- *$t_{nom}(P) = \max_{a \in P} L_{nom}(a)$, and*
- *$t_{wc}(P) = \max_{a \in P} L_{wc}(a)$.*

**Proof.** This follows from Lemma 3 and from the construction of the labels $L_{nom}$ in Algorithm 1 as the sum of the travel time $\pi_j - s$ until node $j$ and the shortest path travel time $K_{nom}(j)$ from $j$ to $v$. ◀

As a conclusion, we obtain the following theorem.

▶ **Theorem 5.** *The bicriteria recoverable robust path problem corresponds to a bicriteria bottleneck shortest path problem in the EAN with labels $L$.*

It is folklore that the single-criteria bottleneck shortest path problem can be solved in linear time on directed acyclic graphs. The Pareto front of bicriteria bottleneck shortest path problems can be found in $O(|\mathcal{A}|^2)$ by a simple $\varepsilon$-constraint method which enumerates all possible values of the first objective function, deletes edges whose labels exceed the given value, and finds a bottleneck shortest path with respect to the second criterion in the remaining graph (compare [12]).

▶ **Lemma 6.** *Algorithm 1 determines the labels $L$ in time $O(|\mathcal{A}| \cdot |\mathcal{U}|)$.*

**Proof.** The initialization takes time $O(|\mathcal{A}|\cdot|\mathcal{U}|)$. Since we can assume that $\mathcal{N}$ is topologically sorted, shortest paths from a node to all other nodes can be found in time $O(|\mathcal{A}|)$. Hence, step 7 takes time $O(|\mathcal{A}|)$. For every $\xi \in \mathcal{U}$, determining $\mathcal{A}^\xi$ is in $O(|\mathcal{A}|)$. Since step 10 again is a shortest path calculation in a topologically sorted network and the operations in the loop over all $(i,j) \in \bigcup_{\xi\in\mathcal{U}} A^\xi$ take constant time, steps 8-15 can be executed in time $O(|\mathcal{A}| \cdot |\mathcal{U}|)$. ◀

## 3.2 Single-Criteria Versions of Recoverable Robustness

To calculate the Pareto front of the bicriteria recoverable robust path problem with finite uncertainty set, we can use the approach as sketched in the previous section. However, we are also interested in two single-criteria versions of the problem. In particular, results of versions with single objective values can be much easier compared for sets of instances.

▶ **Problem 2.** *Worst-case optimal recoverable robust paths*
Find a recoverable robust path $P$ from $u$ to $v$ in $\mathcal{N}$ such that

- the nominal quality of $P$ is smaller or equal than a given nominal quality bound $T_{nom}$,
- $P$ minimizes $t_{wc}(P)$.

▶ **Problem 3.** *Nominally optimal recoverable robust paths*
Find a recoverable robust path $P$ from $u$ to $v$ in $\mathcal{N}$ such that

- the worst-case quality of $P$ is smaller or equal than a given worst-case quality bound $T_{wc}$
- $P$ minimizes $t_{nom}(P)$.

Algorithm 2 describes how to compute worst-case optimal recoverable robust paths. The pseudo-code for an analogous algorithm to compute nominally optimal recoverable robust path, Algorithm 3, is provided in the Appendix.

**Algorithm 2** Worst-case optimal recoverable robust path

```
Require: Network $\mathcal{N} = (\mathcal{E}, \mathcal{A})$, labels $L$, nominal quality bound $T_{nom}$, origin event $u$,
    destination event $v$.
Ensure: Path $P$ which is optimal for Problem 2 (if existing).
 1: for $a \in \mathcal{A}$ do
 2:     if $L_{nom}(a) > T_{nom}$ then
 3:         Remove $a$ from $\mathcal{A}$.
 4:     end if
 5: end for
 6: Find a bottleneck shortest path $P_{wc}$ in $\mathcal{N}$ according to labels $L_{wc}$.
 7: if there is no such path with length $< \infty$ then
 8:     return There is no recoverable robust path.
 9: else
10:     Let $(i,j)$ be the last arc on $P_{wc} \cap \bigcup_{\xi\in\mathcal{U}} \mathcal{A}^\xi$.
11:     Denote by $P^1(j)$ the path $P_{wc}$ until node $j$.
12:     Find a shortest path $P^2(j)$ in $\mathcal{N}$ from $j$ to $v$.
13:     return $P := P^1(j) \cup P^2(j)$, $t_{nom}(P) := \max_{a\in P} L_{nom}(a)$, $t_{wc}(P) := \max_{a\in P} L_{wc}(a)$

14: end if
```

**Table 1** Characteristics of the used event activity network and test queries.

| characteristic | event activity network |
|---|---|
| # trains | 38,495 |
| # events | 2,015,664 |
| # stations | 8,857 |
| # transfer activities | 19,869,867 |
| aver. nominal travel time | 398 min |
| aver. # transfers per query | 3.3 |

▶ **Lemma 7.** *Algorithm 2 and Algorithm 3 are correct.*

**Proof.** Let $P$ be the path returned by Algorithm 2 or Algorithm 3. Then, due to the construction of $P$ in step 13 of each algorithm, the assumptions of Lemma 4 are fulfilled, i.e.,

- since $\max_{a \in P} L_{wc}(a) < \infty$, $P$ is recoverable robust,
- $t_{nom}(P) = \max_{a \in P} L_{nom}(a)$, and
- $t_{wc}(P) = \max_{a \in P} L_{wc}(a)$.

Since there is no arc $a$ with $L_{nom} > T_{nom}$ (or $L_{wc} > T_{wc}$, respectively) we have that $t_{nom}(P) \leq T_{nom}$ (or ( $t_{wc}(P) \leq T_{wc}$, respectively). Furthermore, for any other path $P'$ we have that

$$t_{wc}(P') = \max_{a \in P'} L_{wc}(a) \geq \max_{a \in P} L_{wc}(a) = t_{wc}(P)$$

(or $t_{nom}(P') = \max_{a \in P'} L_{nom}(a) \geq \max_{a \in P} L_{nom}(a) = t_{nom}(P)$, respectively). ◀

## 4 Experimental Results

### 4.1 Test Instances

The basis for our computational study is the German train schedule of February 1, 2013 from which we created an event-activity network. We generated transfer activities between pairs of trains at the same station provided that the departing train is scheduled to depart not later than 60 minutes after the planned arrival time of the feeding train. In addition, since some train lines operate only every two hours or irregularly, we add further transfer arcs. Namely, for each arrival event at some station $s$, we also create a transfer arc to those departure events which exceed the time bound of 60 minutes but provide the very next opportunity to get to a neighboring station. The main characteristics of the resulting network are shown in Table 1. To study the robustness of passenger paths, queries should not be too easy. For example, we are not interested in paths which do not require any transfer. Therefore, we decided to generate 1000 relatively difficult queries as follows. For each query, origin and destination are chosen uniformly at random from a set of the 3549 most important stations in Germany (this choice of stations has been provided by Deutsche Bahn AG). Such a pair of origin and destination stations is only accepted if the air distance between them is at least 200km and if the shortest travel route between them requires at least one transfer. The desired start time is uniquely set to 8:00am. The resulting set of queries has an average nominal travel time of 398 minutes and 3.3 transfers per query.

## 4.2 Generating Information Scenarios

A *delay scenario* $d \in \mathbb{N}_0^{\mathcal{A}_{drive} \cup \mathcal{A}_{wait}}$ specifies a delay on each driving and waiting activity. To generate a delay scenario, we first choose the revealing time of the scenario. Afterwards, we decide for each driving and waiting activity whether it shall receive a source delay or not. We use a parameter $p \in (0, 1)$ specifying the probability that a train receives a source delay. This parameter $p$ can be chosen depending on the level of robustness one wants to achieve.

If a train shall be source-delayed, we select one of its driving or waiting activities uniformly at random from those which are scheduled after the revealing time of the scenario and choose the source delay for this activity uniformly at random among 10, 15, 20, 25, and 30 minutes. The source delays on all other activities are set to 0. For simplicity, we assume that trains receive source delays independently from each other.

We use the following basic *delay propagation rule* in order to compute how delays spread along driving, waiting and maintained transfer activities: $\pi(d)$ denotes the timetable adapted to delay scenario $d$. If the start event of an activity $a = (i, j)$ is delayed, also its end event $j$ will be delayed, where the delay can be reduced by the slack time $b_a$. I.e. we require $\pi(d) \geq \pi$ and

$$\pi_j(d) \geq \pi_i(d) + l_a + d_a \tag{1}$$

for all activities $a = (i, j) \in \mathcal{A}_{wait} \cup \mathcal{A}_{drive}$. For transfer activities equation (1) does not necessarily hold. Motivated by real-world decision systems of rail operators, we assume that the decision whether a transfer is actively maintained or not is specified by a *fixed waiting time rule*: Given a number $wt_a \in \mathbb{N}$ for every transfer activity, the transfer is actively maintained if the departing train has to wait at most $wt_a$ minutes compared to its original schedule. If transfer $a$ is actively maintained, we require that (1) holds for it. However, if for a transfer activity $a = (i, j)$ (1) holds due to some earlier delay on the train corresponding to $j$, $a$ is maintained, even if $\pi_j(d) - \pi_j > wt_a$. Hence, every delay $d$ induces a new set of transfer activities which is denoted as $\mathcal{A}_{transfer}(d)$. Given these waiting time rules for a given delay scenario $d$ we can propagate the delay through the network along the activities in $\mathcal{A}_{drive} \cup \mathcal{A}_{wait} \cup \mathcal{A}_{trans}(d)$ and, thus, calculate the corresponding adapted timetable according to the following propagation rule:

$$\pi_j(d) = \max\left\{\pi_j, \max_{i:(i,j)\in\mathcal{A};\ \pi_i(d)+l_{ij} \leq \pi_j + wt_{ij}} \{\pi_i(d) + l_{ij} + d_{ij}\}\right\} \tag{2}$$

where we set $wt_a = \infty\ \forall a \in \mathcal{A}_{wait} \cup \mathcal{A}_{drive}$ and $d_a = 0\ \forall a \in \mathcal{A}_{trans}$. The concrete waiting time rule used in our experiments is that high speed trains (like Intercity Express ICE, Intercity IC, and Eurocity EC) wait for each other at most three minutes, whereas trains of other train categories do not wait. Note that delay propagation can be done in time $O(|\mathcal{A}|)$. The uncertainty sets used in our experiments contain a number $k$ of independent scenarios generated as described above.

## 4.3 Environment

All experiments were run on a PC (Intel(R) Xeon(R), 2.93GHz, 4MB cache, 47GB main memory under Ubuntu Linux version 12.04 LTS). Only one core has been used by our program. Our code is written in C++ and has been compiled with g++ 4.6.3 and compile option -O3.

## 4.4 Experiments

The purpose of this study is to evaluate the potential of recoverable robust paths as an alternative timetable information method in pretrip planning. A standard way of doing timetable information is to search for a path with minimum travel time as primary objective and with minimum transfers as a secondary one. We take this kind of standard search as the baseline of our comparisons.

**Experiment 1: What is the effect of delays on the paths of the standard search?** We perform the following evaluation. Suppose that $P$ is a given path. For each delay scenario, we determine the first event after the scenario's revealing time. We assume that the passenger can adjust his/her path to the delay scenario at this point and therefore compute the earliest arrival time at the destination under these conditions. The worst-case arrival time over all scenarios is the value we are interested in. To each of our 1000 test queries we applied the same set of 100 delay scenarios with parameter $p = 0.20$. We observe that on average the worst-case travel time is 450 minutes, i.e., 13% larger than the planned one. The absolute difference is 52 minutes on average.

**Experiment 2: What is the price of a worst-case optimal recoverable robust path in comparison with a standard path?** Using the same 100 delay scenarios as for Experiment 1, we are interested in two quantities, namely the nominal travel time and the worst-case travel time of a worst-case recoverable robust path. We upper bounded the nominal arrival time of a recoverable robust path by 150% of the fastest nominal path. Among all paths satisfying this bound we minimized the worst-case arrival time over all scenarios. Our computational results show that for all 1000 queries but two cases there exists a recoverable robust path. An interesting observation is that 34.2% of all standard paths are already the worst-case optimal recoverable robust paths. However, in 27% of the queries the worst-case arrival time is improved in comparison with the standard path. If there is an improvement, the reduction is 29 minutes on average, but the maximum observed difference is 220 minutes. The histogram in Figure 1 gives a more detailed picture. It shows how often a saving of $x$ minutes in the worst-case scenario can be achieved by choosing a recoverable robust path. The price a passenger has to pay if he/she chooses a recoverable robust path is a slight average increase in nominal travel time to 407 minutes, i.e., about just 9 minutes more than for the standard search.

In Figure 2, we show box-and-whisker plots for the distributions of travel times for five algorithmic variants. The data is based on our test set of 1000 queries, each evaluated for 100 delay scenarios generated with parameter $p = 0.2$ for the probability that a train will be delayed by a source delay.

Recall that StNom and StWC stand for the nominal and worst-case travel time in minutes of the standard search, while RRNom and RRWC denote the nominal and worst-case travel time for worst-case optimal recoverable robust paths, respectively. Finally, SRNom gives the nominal travel time for strictly robust paths.

**Experiment 3: What is the influence of parameter $p$, initially chosen as $p = 0.2$?** Recall that parameter $p$ specifies the probability that a train will be delayed by a source delay. To quantify the sensitivity of the different solution methods on the chosen uncertainty set, we redo the previous two experiments with $p = 0.1$ and $p = 0.15$. Figure 3 (left) and Table 2 summarize our findings and show the average nominal (StNom) and worst case travel time (StWC) in minutes for the standard search and the nominal (RRNom) and worst-case time (RRWC) for the optimal recoverable robust paths, respectively. If the probability parameter $p$ increases, we observe a slight increase of average worst case travel times (what

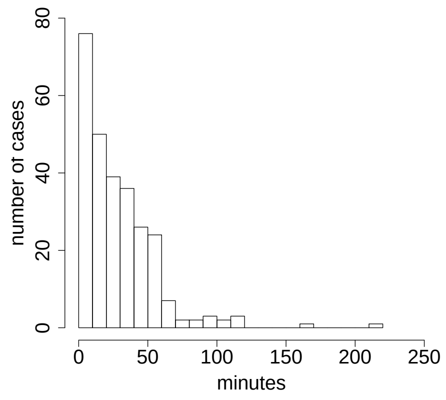


**Figure 1** This histogram shows the number of cases where with respect to the worst-case scenario we can save $x$ minutes by choosing a worst-case optimal recoverable robust path instead of the standard path.

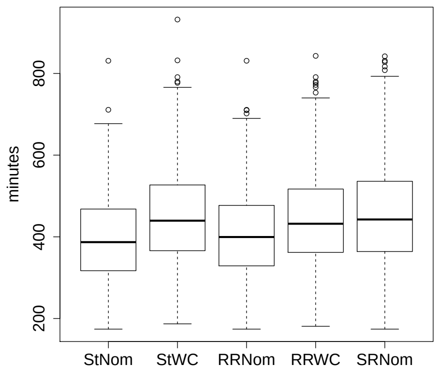
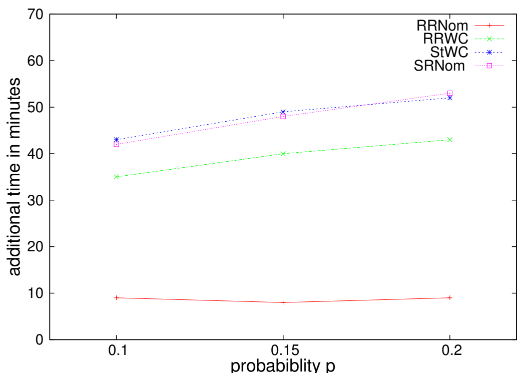


**Figure 2** Box-and-whisker plots for the travel time distributions of several algorithmic variants.



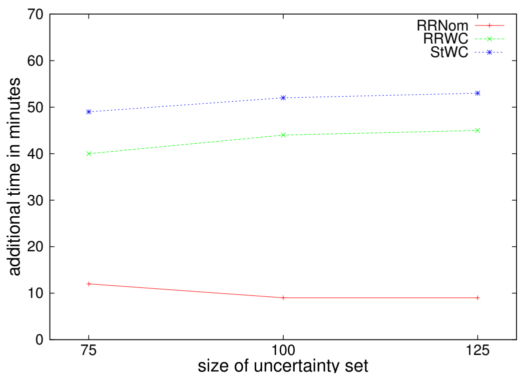


**Figure 3** Additional travel time over the baseline of the standard path in minutes for different values of probability paramter $p$ (left) and different number of scenarios (right). The average nominal travel time for standard paths is 498 minutes.

should be expected), whereas the nominal travel time of recoverable robust paths is almost unchanged. We conclude that $p = 0.2$ might be preferable since it provides recoverability for the more severe scenarios at no price with respect to nominal travel time.

Table 2 shows the raw data from which Figure 3 (left) has been derived.

**Experiment 4: Comparison with strictly robust paths.** Using the same uncertainty set as in the previous experiments, we computed the set of transfer activities which break at least once. We marked these arcs as forbidden, and rerun shortest path queries on the resulting even-activity network. Paths in this network are considered as strictly robust since no transfer will ever break. The average nominal travel time if we look for the fastest strictly robust path (SRNom) is 451 minutes for the uncertainty set with $p = 0.2$ (see also Figure 3 (left) and the last row of Table 2. Hence, the average nominal travel time of these paths is not better than the average worst-case time for standard paths. In full agreement with previous studies [17, 18], strictly robust paths turn out to be too conservative.

**Table 2** Comparison of standard and robust solutions: Average travel time in minutes for $k = |U| = 100$ scenarios.

| | $p = 0.10$ | $p = 0.15$ | $p = 0.20$ |
|---|---|---|---|
| StNom | 398 | 398 | 398 |
| StWC | 441 | 447 | 450 |
| RRNom | 407 | 406 | 407 |
| RRWC | 433 | 438 | 442 |
| SRNom | 440 | 446 | 451 |

**Table 3** Comparison of standard and robust solutions for different sizes $k$ of the uncertainty set: Average travel time in minutes for $p = 0.20$.

| | $k = 75$ | $k = 100$ | $k = 125$ |
|---|---|---|---|
| StNom | 398 | 398 | 398 |
| StWC | 447 | 450 | 451 |
| RRNom | 410 | 407 | 407 |
| RRWC | 438 | 442 | 443 |

**Experiment 5: To which extent do our observations depend on the size of the scenario set?** All previous experiments have been run with 100 different delay scenarios. The parameter $k = |\mathcal{U}|$ has been chosen as a pragmatic compromise between efficiency (the computational effort scales linearly with $k$) and the degree of robustness we want to guarantee. Obviously, the more different scenarios we use, the higher the level of robustness we can achieve. Therefore, we fixed the parameter $p = 0.20$ but varied $k \in \{75, 100, 125\}$. Table 3 shows the average travel times in minutes for these variants, and Figure 3 (right) displays the additional travel time over the baseline of the standard path in minutes. It is interesting to observe that the average worst-case travel times depend only marginally on the parameter $k$ in the chosen range. As expected, there is a slight increase of a few minutes on worst-case travel time when we increase $k$. At the same time, the average nominal travel time for recoverable robust paths does not increase. Further experiments will be needed to see whether this trend will be confirmed if $k$ is chosen in an even wider range.

**Practicality of our approach.** For the purpose of this study, we have merely implemented a first prototype without much emphasis on performance issues. Our running times are several minutes per query which is clearly impractical. The main bottleneck is the computation of labels which grows linearly with the number of used scenarios. However, the most expensive part, namely the loop of lines 8-15, could be run in parallel. Thus, using massive parallelization and further speed-up techniques, we see a clear perspective that the computation time for a recoverable robust path can be brought down to a few seconds.

## 5 Conclusion and Further Research

In this work we introduced the concept of time-dependent recoverable-robust paths within the framework of timetable information. We showed that the resulting bicriteria problem can be solved in polynomial time using a label-setting algorithm, and a subsequent bottleneck shortest path calculation. The proposed concept and algorithm was experimentally evaluated on timetable information instances covering the whole German train network (schedule of 2013). While computation times are still too high for practical applications in the current implementation, we may assume that a parallelized algorithm will be sufficiently fast; moreover, as our experiments show that the proposed model has a valuable trade-off between nominal and worst-case travel times, such an algorithm will provide a customer-friendly alternative in practice. Further research includes the comparison of recoverable robust paths to lightly robust paths (see [17, 18]), and the extension of the proposed model to multi-stage robustness where only partial information on the scenario is given at discrete points in time. Also, the evaluation of the computed paths with respect to a set of real delay scenarios is currently being analyzed.

## References

1 A. Ben-Tal, A. Goryashko, E. Guslitzer, and A. Nemirovski. Adjustable robust solutions of uncertain linear programs. *Math. Programming A*, 99:351–376, 2003.

2 A. Ben-Tal and A. Nemirovski. Robust convex optimization. *Mathematics of Operations Research*, 23(4):769–805, 1998.

3 A. Ben-Tal and A. Nemirovski. Robust solutions of linear programming problems contaminated with uncertain data. *Math. Programming A*, 88:411–424, 2000.

4 A. Berger, M. Grimmer, and M. Müller-Hannemann. Fully dynamic speed-up techniques for multi-criteria shortest paths searches in time-dependent networks. In P. Festa, editor, *Proceedings of SEA 2010*, volume 6049 of *LNCS*, pages 35–46. Springer, Heidelberg, 2010.

5 D. Bertsimas and M. Sim. The price of robustness. *Operations Research*, 52(1):35–53, 2004.

6 C. Büsing. Recoverable robust shortest path problems. *Networks*, 59(1):181–189, 2012.

7 C. Büsing, A. M. C. A. Koster, and M. Kutschka. Recoverable robust knapsacks: the discrete scenario case. *Optimization Letters*, 5(3):379–392, 2011.

8 V. Cacchiani, A. Caprara, L. Galli, L. Kroon, G. Maroti, and P. Toth. Railway rolling stock planning: Robustness against large disruptions. *Transportation Science*, 46(2):217–232, May 2012.

9 A. Caprara, L. Galli, S. Stiller, and P. Toth. Recoverable-robust platforming by network buffering. Technical Report ARRIVAL-TR-0157, ARRIVAL Project, 2008.

10 S. Cicerone, G. D'Angelo, G. Di Stefano, D. Frigioni, and A. Navarra. Robust Algorithms and Price of Robustness in Shunting Problems. In *ATMOS 2007*, 2007.

11 S. Cicerone, G. D'Angelo, G. Di Stefano, D. Frigioni, A. Navarra, M. Schachtebeck, and A. Schöbel. Recoverable robustness in shunting and timetabling. In *Robust and Online Large-Scale Optimization*, volume 5868 of *LNCS*, pages 28–60. Springer, Heidelberg, 2009.

12 L. de Lima Pinto, C. T. Bornstein, and N. Maculan. The tricriterion shortest path problem with at least two bottleneck objective functions. *European Journal of Operational Research*, 198:387–391, 2009.

13 J. Dibbelt, Th. Pajor, B. Strasser, and D. Wagner. Intriguingly simple and fast transit routing. In V. Bonifaci, C. Demetrescu, and A. Marchetti-Spaccamela, editors, *Experimental Algorithms*, volume 7933 of *LNCS*, pages 43–54. Springer, Heidelberg, 2013.

14 Y. Disser, M. Müller-Hannemann, and M. Schnee. Multi-criteria shortest paths in time-dependent train networks. In C. C. McGeoch, editor, *WEA 2008*, volume 5038 of *LNCS*, pages 347–361. Springer, Heidelberg, 2008.

15 A.L. Erera, J.C. Morales, and M. Savelsbergh. Robust optimization for empty repositioning problems. *Operations Research*, 57(2):468–483, 2009.

16 M. Fischetti and M. Monaci. Light robustness. In R. K. Ahuja, R.H. Möhring, and C.D. Zaroliagis, editors, *Robust and online large-scale optimization*, volume 5868 of *LNCS*, pages 61–84. Springer, Heidelberg, 2009.

17 M. Goerigk, M. Knoth, M. Müller-Hannemann, M. Schmidt, and A. Schöbel. The price of robustness in timetable information. In *ATMOS 2011*, volume 20 of *OASICS*, pages 76–87. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany, 2011.

18 M. Goerigk, M. Knoth, M. Schmidt, A. Schöbel, and M. Müller-Hannemann. The price of strict and light robustness in timetable information. *Transportation Science*, 2013. To appear.

19 C. Liebchen, M. Lübbecke, R. H. Möhring, and S. Stiller. The concept of recoverable robustness, linear programming recovery, and railway applications. In R. K. Ahuja, R.H. Möhring, and C.D. Zaroliagis, editors, *Robust and online large-scale optimization*, volume 5868 of *LNCS*, pages 1–27. Springer, Heidelberg, 2009.

20 M. Müller-Hannemann and M. Schnee. Efficient timetable information in the presence of delays. In R. Ahuja, R.-H. Möhring, and C. Zaroliagis, editors, *Robust and Online Large-Scale Optimization*, volume 5868 of *LNCS*, pages 249–272. Springer, Heidelberg, 2009.

21 M. Müller-Hannemann, F. Schulz, D. Wagner, and C. Zaroliagis. Timetable information: Models and algorithms. In *Algorithmic Methods for Railway Optimization*, volume 4395 of *LNCS*, pages 67–89. Springer, Heidelberg, 2007.

22 M. Schnee. *Fully realistic multi-criteria timetable information systems.* PhD thesis, Fachbereich Informatik, Technische Universität Darmstadt, 2009. Published in 2010 by Südwestdeutscher Verlag für Hochschulschriften.

23 O. Seref, A. Ravindra, and J. B. Orlin. Incremental network optimization: Theory and algorithms. *Operations Research*, 57(3):586–594, 2009.

24 A.L. Soyster. Convex programming with set-inclusive constraints and applications to inexact linear programming. *Operations Research*, 21:1154–1157, 1973.

25 S. Stiller. *Extending concepts of reliability. Network creation games, real-time scheduling, and robust optimization.* PhD thesis, TU Berlin, 2008.

## A Algorithmic Approach

Algorithm 3 describes how to find nominally optimal recoverable robust paths subject to an upper worst-case quality bound.

**Algorithm 3** Nominally optimal recoverable robust path

```
Require: Network $\mathcal{N} = (\mathcal{E}, \mathcal{A})$, labels $L$, worst-case quality bound $T_{wc}$, origin event $u$,
    destination event $v$.
Ensure: Path $P$ which is optimal for Problem 3 (if existing).
 1: for $a \in \mathcal{A}$ do
 2:     if $L_{wc}(a) > T_{wc}$ then
 3:         Remove $a$ from $\mathcal{A}$.
 4:     end if
 5: end for
 6: Find a bottleneck shortest path $P_{nom}$ in $\mathcal{N}$ according to labels $L_{nom}$.
 7: if there is no such path with length $< \infty$ then
 8:     return There is no recoverable robust path.
 9: else
10:     Let $(i, j)$ be the last arc on $P_{nom} \cap \bigcup_{\xi \in \mathcal{U}} \mathcal{A}^{\xi}$.
11:     Denote by $P^1(j)$ the path $P_{nom}$ until node $j$.
12:     Find a shortest path $P^2(j)$ in $\mathcal{N}$ from $j$ to $v$.
13:     return $P := P^1(j) \cup P^2(j)$, $t_{nom}(P) := \max_{a \in P} L_{nom}(a)$, $t_{wc}(P) := \max_{a \in P} L_{wc}(a)$
14: end if
```

# Is Timetabling Routing Always Reliable for Public Transport?

**Donatella Firmani[1], Giuseppe F. Italiano[1], Luigi Laura[2], and Federico Santaroni[1]**

1 **Department of Civil Engineering and Computer Science Engineering University of Rome "Tor Vergata", Rome, Italy**
`firmani@ing.uniroma2.it, italiano@disp.uniroma2.it, santaroni@ing.uniroma2.it`

2 **Department of Computer, Control, and Management Engineering and Research Centre for Transport and Logistics – Sapienza University of Rome, Italy**
`laura@dis.uniroma1.it`

## Abstract

Current route planning algorithms for public transport networks are mostly based on timetable information only, i.e., they compute shortest routes under the assumption that all transit vehicles (e.g., buses, subway trains) will incur in no delays throughout their trips. Unfortunately, unavoidable and unexpected delays often prevent transit vehicles to respect their originally planned schedule. In this paper, we try to measure empirically the quality of the solutions offered by timetabling routing in a real public transport network, where unpredictable delays may happen with a certain frequency, such as the public transport network of the metropolitan area of Rome. To accomplish this task, we take the time estimates required for trips provided by a timetabling-based route planner (such as Google Transit) and compare them against the times taken by the trips according to the actual tracking of transit vehicles in the transport network, measured through the GPS data made available by the transit agency. In our experiments, the movement of transit vehicles was only mildly correlated to the timetable, giving strong evidence that in such a case timetabled routing may fail to deliver optimal or even high-quality solutions.

**1998 ACM Subject Classification** F.2 Analysis of Algorithms and Problem Complexity, F.2.2 Nonnumerical Algorithms and Problems

**Keywords and phrases** Shortest Path Problems, Route Planning, Timetable-based Routing, Public Transport Networks

**Digital Object Identifier** 10.4230/OASIcs.ATMOS.2013.15

## 1 Introduction

In the last years we have witnessed an explosion of exciting research on point-to-point shortest path algorithms for road networks, motivated by the widespread use of navigation software. Many new algorithmic techniques have been introduced, including hierarchical approaches (e.g., contraction hierarchies) [15, 26], reach-based approaches [18, 19], transit node routing [6], and hub-based labeling algorithms [1]. (Delling et al. [11] gives a more detailed overview of the literature.) The algorithms proposed in the literature are of great practical value, as on average they are several orders of magnitude faster than Dijkstra's algorithm, which is too slow for large-scale road networks: on very large road networks, such as the entire Western Europe or North America, the fastest algorithms are able to compute point-to-point distances in few microseconds on high-performance computing platforms and in hundred milliseconds on mobile devices (see e.g., [17]). Computing the actual shortest paths (not only distances) requires slightly more time (i.e., few order of magnitudes), but it is still very fast in practice. We remark that this algorithmic work had truly a big practical impact on navigation systems: some of the ideas introduced in the scientific literature are currently used by Apple, Bing and Google Maps. Furthermore, this research on point-to-point shortest path algorithms generated not only results of practical value, but also deep theoretical questions that gave rise to several exciting results: Abraham et al. [2] gave theoretical justifications of the practical efficiency of some of those approaches under the assumption of low highway dimension (HD) of the input graph, which is believed to be true for road networks, and even showed some amazing relationships to VC dimension [1].

Although most algorithmic techniques designed for road networks can be immediately transferred to public transport networks, unfortunately their adaptation to this case is harder than expected, and they fail to yield comparable speed-ups [5, 14]. One of the reasons, as explained in the excellent work of Bast [4], is that most public transportation networks, like bus-only networks in big metropolitan areas, are far more complex than other types of transportation networks, such as road networks: indeed, public transport networks are known to be less hierarchically structured and are inherently event-based. Thus, it seems that, in order to achieve significant speed-ups on public transport networks, one needs to take into account more sophisticated and larger scale time-dependent models [9, 14, 24, 25] or to develop completely different algorithmic techniques, such as either the transfer patterns introduced by Bast et al. [5], the approach based on dynamic programming by Delling et al. [10] or the connection scan by Dibbelt et al. [12].

Current route planning algorithms for public transport networks are mostly based on timetable information, i.e., they compute shortest routes under the assumption that all transit vehicles (e.g., buses, subway trains) will start their trip exactly at the planned time and that they will incur in no delays throughout their journey. However, in our daily experience buses often run behind schedule: unavoidable delays occur frequently and for many unplanned reasons, including traffic jams, accidents, road closures, inclement weather, increased ridership, vehicle breakdowns and sometimes even unrealistic scheduling. As a consequence, widely used timetable routing algorithms may suffer from several inaccuracies: the more buses run behind schedule, the more is likely that routing methods based on timetabling will not be able to estimate correctly the waiting times at bus stops, thus failing to deliver optimal solutions, i.e., the actual shortest routes. Indeed, in the recent past, a lot of effort has been put in developing either robust models able to efficiently cope with delays and cancellation events [8, 13, 16, 7], or dynamic delay propagation models for the design of robust timetables and the evaluation of dispatching proposals [23]. These approaches yield interesting insights into the robustness of the solutions offered against small fluctuations.

In this framework, it seems quite natural to ask how much timetabling-based routing methods are effectively able to deliver optimal solutions on actual public transport networks. To address this complex issue, in this paper we try to measure the quality of the solutions offered by timetabling routing in the public transport network of the metropolitan area of a big city, where unpredictable delays, unplanned disruptions or unexpected events seem to happen with a certain frequency. As a first step, we consider the public transport network of Rome: we believe that fluctuations on the transit schedule are not limited to this case, but they happen often in many other urban areas worldwide. In more detail, we performed the following experiment. On a given day, we submitted to Google Transit, the well known public transport route planning tool integrated in Google Maps, many queries having origin and destination in the metropolitan area of Rome: in this case, the journeys computed by

Google Transit are based on the timetabling data provided by the transit agency of Rome[1]. Besides its origin and destination, each query $q_i$ is characterized by the starting time $\tau_i$ from the origin. For each query $q_i$, on the same day we followed precisely the journeys suggested by Google Transit, starting at time $\tau_i$, by tracking in real time the movement of transit vehicles in the transport network through the GPS data made available by the very same transit agency. In order to do that efficiently, we collected the GPS data on the geo-location of all vehicles on the very same day, by submitting queries every minute to the transit agency of Rome [28]. With all the data obtained, we built a simulator capable of following precisely each journey on that given day, according to the GPS tracking of transit vehicles in the transport network. Finally, we computed the actual total time required by each journey in our simulator and compared it against its original estimate given by Google Transit. We believe that the simulator built for this experiment was not only instrumental for its success, but it can also be of independent interest for other investigations in a public transport network.

Our experimental analysis shows that in the public transport network considered the movement of transit vehicles was only mildly correlated to the original timetable. In such a scenario, timetabled-based routing methods suffer from many inaccuracies, as they are based on incorrect estimations of the waiting/transfer times at transit stops, and thus they might fail to deliver an optimal or even high-quality solution. In this case, in order to compute the truly best possible routes (for instance, shortest time routes), it seems that we have to overcome the inherent oversights of timetable routing: toward this end, we advocate the need to design new route planning algorithms which are capable of exploiting the real-time information about the geo-location of buses made available by many transit authorities.

## 2 Preliminaries

In the following we introduce some basic terminology which will be useful throughout the paper. Our public transport networks consist of a set of *stops*, a set of *hops* and a set of *footpaths*:

- A *stop* corresponds to a location in the network where passengers may either enter or exit a transit vehicle (such as a bus stop or a subway station).
- A *hop* is a connection between two adjacent stops and models a vehicle departing from stop $u$ and arriving at stop $v$ without intermediate stops in between.
- A *trip* consists of a sequence of consecutive hops operated by the same transit vehicle. Trips can be grouped into *lines*, serving the exact same sequence of consecutive hops.
- A *footpath* enables walking transfers between nearby stops. Each footpath consists of two stops and an associated (constant) walking time between the two stops.
- A *journey* connects a source stop $s$ and a target stop $t$, and consists of a sequence of trips and footpaths in the order of travel. Each trip in the journey is associated with two stops, corresponding to the pick-up and drop-off points.

[1] Roma Servizi per la Mobilità [28].

## 3 Experimental Setup

### 3.1 Experiments

In our experiments, we considered the public transport network of Rome, which consists of 309 bus lines and 3 subway lines, with a total of 7,092 stops (7,037 bus stops and 55 subway stops). We generated random queries, where each query $q_i$ consisted of a triple $\langle s_i, t_i, \tau_i \rangle$:

- $s_i$ is the start stop;
- $t_i$ is the target stop;
- $\tau_i$ is the time of the departure from the start stop.

Our experiments were carried out as follows. Each start and target stop $s_i$ and $t_i$ was generated uniformly at random in the metropolitan area of Rome, while the departure time $\tau_i$ was chosen uniformly at random between 7:00am and 9:00pm. We selected Thursday June 6, 2013 as a day for our experiments, and in this day we did not observe any particular deviation form the typical delays in the trips. We submitted each query $q_i$ to Google Transit on the very same day (June 6, 2013), and collected all the journeys suggested in return to the query and their predicted traveling times. In the vast majority of cases, Google Transit returns 4 journeys, but there were queries that returned less than 4 public transit journeys; this might happen, for instance, when one of the journeys returned is a footpath. This produced a total of $4,018$ journeys. Note that, since Google Transit is based on the timetabled data provided by the transit agency of Rome, the predicted traveling time of each journey is computed according to the timetable.

We next tried to measure empirically the actual time required by each such journey in the real public transport network. We performed this as follows. On June 6, 2013 we submitted queries every minute to the transit agency of Rome [28], in order to obtain (from GPS data) the instantaneous geo-location of all vehicles in the network. Given that stream of GPS data, we built a simulation system capable of following precisely each journey from a given starting time, according to the GPS tracking of transit vehicles in the transport network. We describe this process in more detail in Section 3.2. Finally, we computed the actual total time required by each route in our simulator and compared it against its original estimate given by Google Transit.

### 3.2 Simulation system

Our system makes it possible to simulate closely the experience of a user traveling according to each input journey, after leaving the origin at the corresponding time. For each trip in the journey, the pick-up and drop-off times are computed according to the position of transit vehicles in the public transport network. A user can be picked-up or dropped-off either earlier or later than originally scheduled, and if a delayed transit vehicle misses a connection then the next trip of the same line is chosen. To obtain the real-time position of ground vehicles (such as buses, trains or trams) we used streamed GPS data, while for trips which do not provide vehicle live positions (such as saubway train trips) we employed their original estimate given by Google Transit. This allows us to follow input journey containing both ground and underground trips as well. We remark that all of the journeys produced in our experiments contained at least one trip operated by ground vehicles. Finally, we used Google Maps to compute the times needed by footpaths.

## 4 Experimental Results

In this section we report the results of our experiments. We compare the *estimated time* $t_e(j)$ required by each journey $j$ according to the timetable (as reported by Google Transit), and its *actual time* $t_a(j)$ computed from the vehicle real-time positions given by the stream of GPS data (as contained in our simulation system). More specifically, we define the *error coefficient* of journey $j$ to be $t_a(j)/t_e(j)$. Note that the error coefficient measures the distance between the time predicted by timetabling routing and the actual time that journey $j$ will incur in reality. It will be equal to 1 whenever the actual journey will be in perfect agreement with the times predicted by timetabling routing. It will be larger than 1 whenever the actual journey will be slower than what was predicted by timetabling routing (increased waiting times at a bus stop for a delayed connection). It will be smaller than 1 whenever the actual journey will be faster than what was predicted by timetabling routing (smaller waiting times at a bus stop, which can happen in the case a previous connection, which was infeasible by timetabling, was delayed and can become a viable option in the actual journey). Obviously, the more the error coefficient will deviate substantially from 1 (especially in the case where it is larger than 1), the less accurate will be the time estimations of timetabling routing and the more likely is that timetabling routing will fail to compute the shortest journeys.

### 4.1 Measured error coefficients

To report the distribution of the error coefficients as a function of the journey time, we proceed as follows. For each journey $j$, the journey time is taken as the estimated time $t_e(j)$ according to the timetable. Since there can be multiple journeys sharing the same value of $t_e(j)$, we group those journeys into *time slots* within a 3-minute resolution. More formally, we measure $t_e(j)$ in minutes and the $k$-th time slot $\sigma_k$ contains all journeys $j$ such that $t_e(j) \in [3k, 3(k+1)]$. For each time slot, we look at the proximity of the obtained error coefficient distribution to the constant 1, which represents the ideal scenario where the times of actual journeys are in perfect agreement with the times predicted by timetabling routing. To this end, we compute the metrics below:

- **Average.** We measure the average of the error coefficient in each time slot.
- **Percentiles.** Analogously, for each time slot $\sigma$, we measure the 10th percentile and the 90th percentile of the error coefficients.
- **Minimum-Maximum.** Finally, we measure $\min_{j\in\sigma}\{\frac{t_a(j)}{t_e(j)}\}$ and $\max_{j\in\sigma}\{\frac{t_a(j)}{t_e(j)}\}$

We define $t_e(\sigma_k) = 3k + 1.5$ and plot both the evolution of these statistics and the error coefficient, as functions of $t_e$. This also enable us to distinguish between *short distance* journeys, i.e., journeys $j$ with $t_e(j)$ smaller than 30 minutes, *medium distance* journeys, i.e., journeys $j$ with $t_e(j)$ between and 30 and 60 minutes, and *long distance* journeys, i.e., journeys taking more than 60 minutes.

Figure 1 plots the error coefficient for each journey and illustrates the average of the error coefficients for each time slot obtained in our experiments. Note that the error coefficients fluctuate wildly, ranging from 0.15 to 4.44, and the reader may ask how actual trips with extremely small or extremely high error coefficients look like. To this end, we provide more details on two extreme cases, which are a short journey with minimum error coefficient and a long journey with maximum error coefficient, denoted by $j_m$ and $j_M$ respectively:

- $j_m$ consists of a single short distance trip, $t_a(j_m) = 2$ minutes, $t_e(j_m) = 13$ minutes and error coefficient $\approx 0.15$;
- $j_M$ consists of 3 short distance trips and 1 medium distance trip, $t_a(j_M) = 3$ hours and 49 minutes, $t_e(j_M) = 1$ hour and 24 minutes and error coefficient $\approx 2.72$.

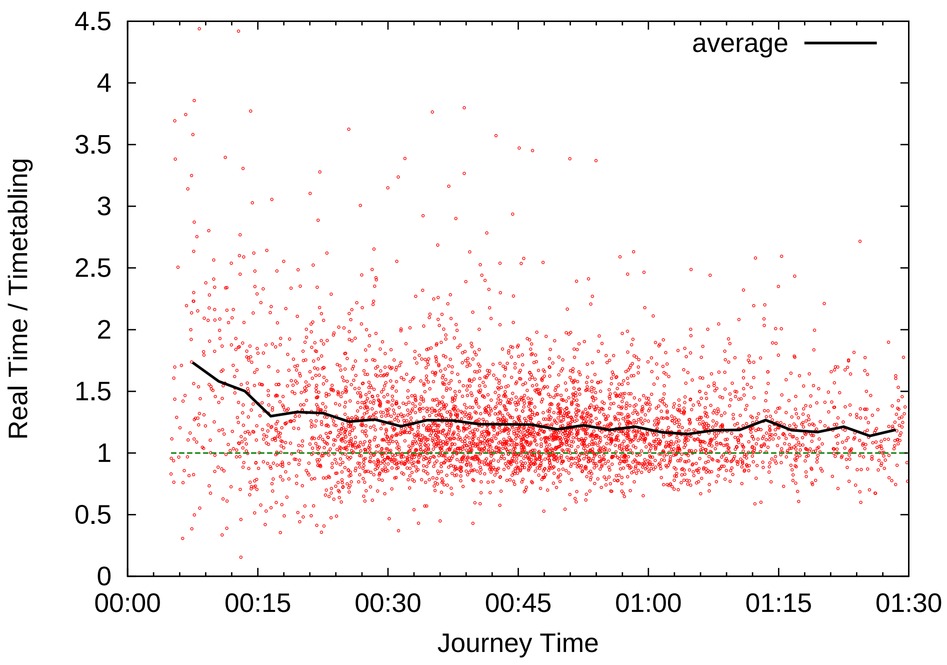


**Figure 1** Distribution of the error coefficients as a function of the journey times (better viewed in color).

The short journey connects two stops which are rather close to each other, and only require a 1-minute bus trip: in this case, the discrepancy between the estimated and the actual travel time is induced by the waiting time at the bus stop. The long journey connects two stops which are rather far away: the journey itself consists of four trips (three short distance and one long distance trip), operated by ground vehicles through intense traffic areas. This results in moderate delays on the short distance trips and a much higher delay on the medium distance trip due to intense traffic.

While high fluctuations are possible, the average error coefficient lies in the interval $[1.13, 1.73]$, which implies that on the average the actual journey times are between 13% and 73% slower than the times used by timetabling routing! In detail, the average error coefficient falls between 1.27 and 1.73 for short journeys, and between 1.13 and 1.26 for long journeys. The fact that the error coefficients appear to be substantially larger for short journeys is not surprising, as short journeys are likely to be more affected (in relative terms) by fluctuations on the schedule. On the other side, larger errors might be less tolerable on short journeys from the users' perspective.

Figure 2 shows the 10th and the 90th percentiles of the distribution of the error coefficients. For the sake of comparison, for each time slot we report also the minimum and the maximum error coefficient. This gives us an interesting insight on a typical user experience: in 80% of the short journeys computed by a timetable-based method, the actual time required ranges from 0.72 to 3.14 of the time estimated with timetabling. Analogously, the same percentage of long journeys takes up to 2 times more than the estimated time. As for the first and last deciles, we observe higher variability in the short journeys rather than in the long journeys. Finally, we observe that 10% of the journeys taking from 15 to 45 minutes are distributed over a long tail in the range $[1.6, 3.8]$. Roughly speaking, 1 such journey out of 10 will take more than twice the scheduled time!

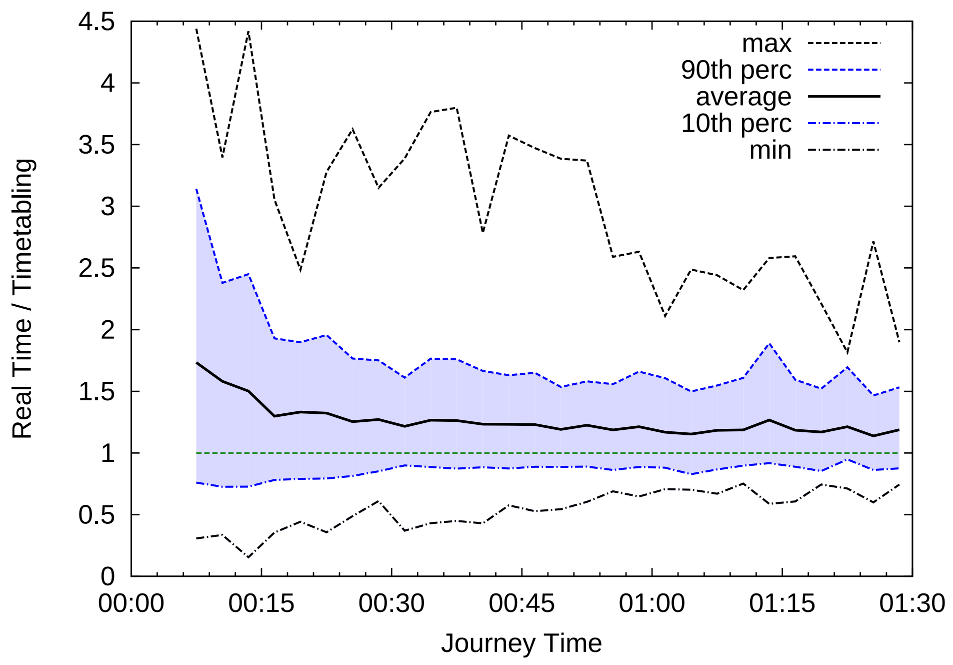


**Figure 2** The 10th and the 90th percentiles of the distribution of the error coefficients (better viewed in color).

It is natural to ask in this scenario whether different discrepancies between the estimated and the actual travel times could be observed under different traffic conditions. As illustrated in Figures 3–5, the distribution of the error coefficients is slightly affected by the different times of the day, which mainly differ for the traffic conditions. This is not surprising, as our queries are generated at random and do not follow the traffic patterns. Since in the morning rush hours there is more traffic towards the city center, while in the evening rush hours the traffic flows out of the city center, only a small percentage of random queries are likely to be affected by those traffic patterns. In the full paper, we will report the result of other experiments that will highlight this phenomenon.

## 4.2 Correlations in ranking

In order to get deeper insights on the differences between the time estimates provided by timetabling and the actual times obtained by tracking transit vehicles in the network, we next investigate the relative rankings of journeys. Namely, for each query we take the four journeys provided by Google Transit and compare their relative rankings in the lists produced by two methods, according to the travel times. If the ranking of the four journeys agree (say, the shortest journey for timetabled routing is also the shortest journey in our real-time simulation with GPS data, the second shortest journey for timetabled routing is also the shortest journey in our real-time simulation, etc...) then there is a strong correlation between the two rankings, independently of the values of the journey times.

To assess the degree of similarity between the two rankings, we use the Kendall Tau coefficient [22]. This is a rank distance metric that counts the number of pairwise disagreements between two ranking lists: the larger the distance, the more dissimilar the two lists

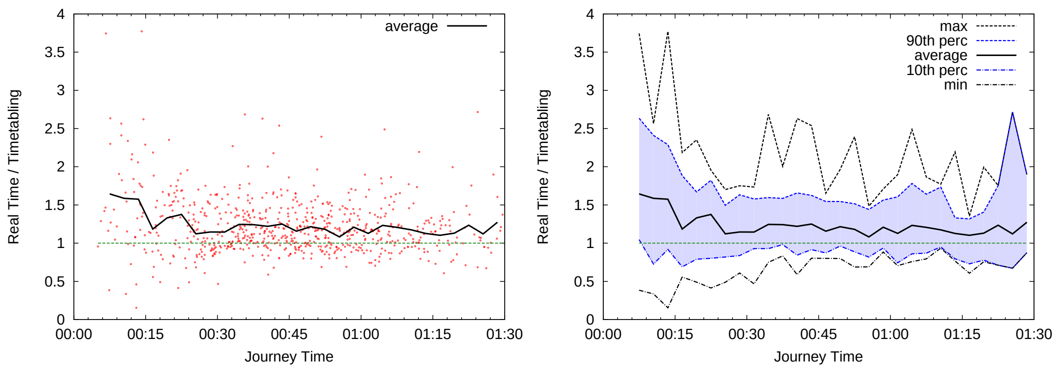


**Figure 3** Distribution of the error coefficients in journeys with time of the departure from 7:30am to 9:30am (better viewed in color).

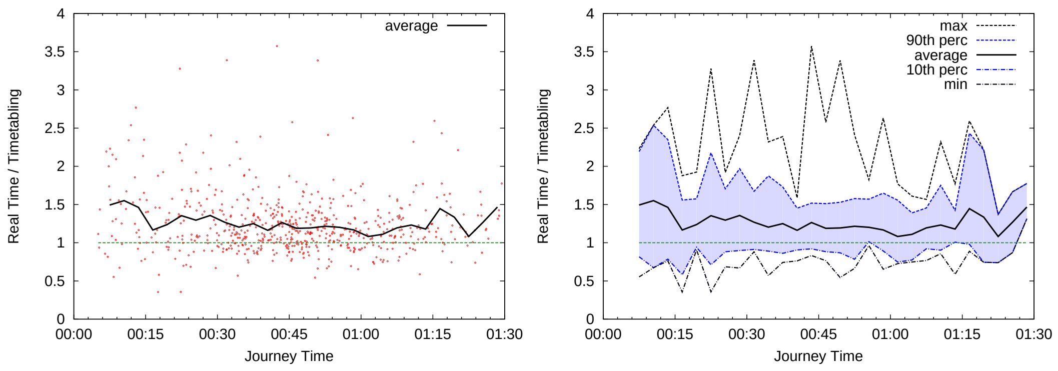


**Figure 4** Distribution of the error coefficients in journeys with time of the departure from 11:30am to 1:30pm (better viewed in color).

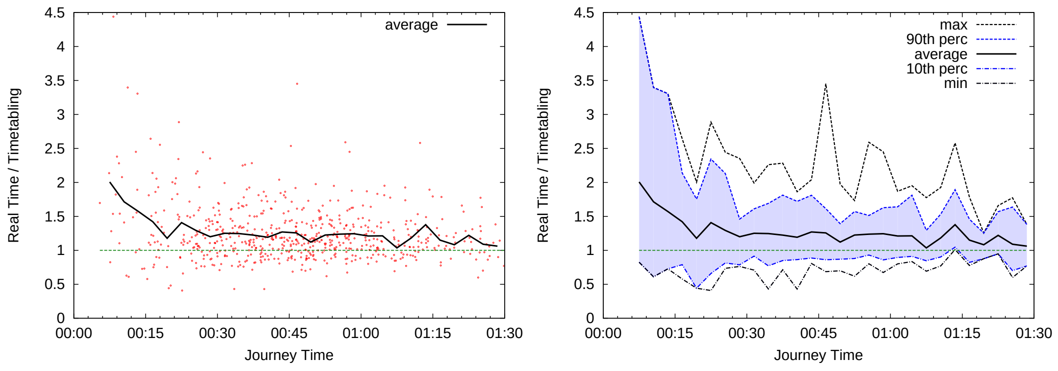


**Figure 5** Distribution of the error coefficients in journeys with time of the departure from 5:00pm to 7:00pm (better viewed in color).

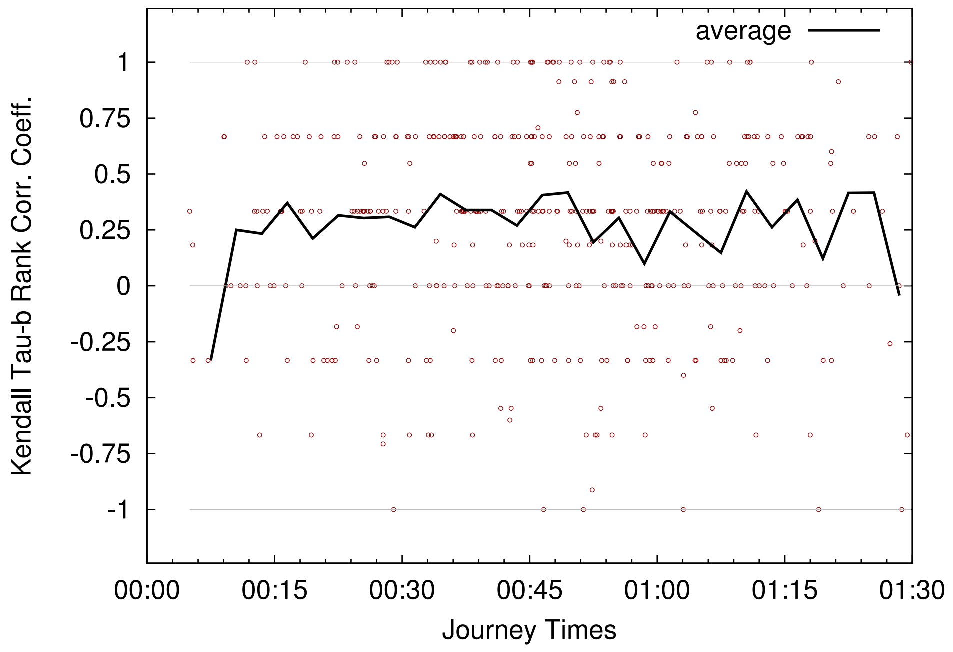


**Figure 6** Kendall Tau-b coefficients for the queries in our experiment (better viewed in color).

are. In particular, we use the Tau-b statistic, which is used when ties exist [3]. The Tau-b coefficient ranges from $-1$ (100% negative association, or perfect inversion) to $+1$ (100% positive association, or perfect agreement): a value of 0 indicates the absence of association (i.e., independence of the two rankings).

Figure 6 shows values of the Kendall Tau-b coefficient for the queries considered in our experiment, plotted against the journey times. As one could expect, in many cases there is a positive correlation between the time estimates provided by timetabling and the actual times obtained by tracking transit vehicles. However, there are also values close to 0, and even worse, there are many negative Tau-b coefficients. The average Tau-b coefficient for each time slot is close to 0.25, which implies only a mildly positive correlation between the two rankings considered. In particular, the average Tau-b coefficient has smaller values for very short journeys and for long journeys: those cases appear to be more vulnerable to fluctuations in the schedule, and thus there seems to be a larger error on the time estimates provided by timetabled routing. In general, the rank correlation analysis given by the Kendall Tau-b statistics shows even more convincing arguments that, according to our experiments in the public transport network considered, timetabled routing fails to deliver optimal or even high-quality solutions.

## 5 Final Remarks

In this paper we measured empirically the quality of the solutions computed by timetabling routing in a real public transport network: for many queries, we compared the time estimate provided by Google Transit with the actual times, computed using the real-time GPS data of the transit vehicles. Our analysis shows that widely used timetable routing algorithms suffer from many inaccuracies, as they are based on incorrect estimations of the waiting/transfer times at transit stops, and thus they might fail to deliver an optimal solution.

The main question that arises naturally in this scenario is how to exploit the real-time information about the geo-location of buses to overcome the inherent oversights of timetable routing and to compute the truly best possible (under several optimization criteria) point-to-point routes, such as shortest routes, routes with minimum number of transfers, etc. As shown recently [20, 21, 27], geo-location data could in fact provide a more accurate and realistic modeling of public transport networks, as they are able to provide better estimates on many variables, such as bus arrival times, the times needed to make a transfer, or the times needed to travel arcs in the transport network. In particular, we expect that this more accurate modeling will make it possible to compute solutions of better quality overall.

Another important issue to investigate is how to compute robust routes, e.g., routes with more backup options (again, based on the current geo-location of buses) and thus less vulnerable to unexpected events. We remark that, whichever is the optimization criterion, route planning with real-time updates on the location of buses appears to be a challenging problem. This is because one has to deal with the sheer size of the input network, augmented with the actual location of buses and combined with a huge bulk of real-time updates, and the fact that such updates provide accurate information only about the past and the current state of the network, while, in order to answer effectively routing queries, one still needs to infer some realistic information about the future. Perhaps, this explains why a solution to these problems has been elusive, despite the fact that geo-location data have been already available for many years.

**Acknowledgements** We wish to thank the transit agency of Rome, Roma Servizi per la Mobilità, and in particular Luca Allulli, Carlo Gentile, Damiano Morosi, and Fabio Schiavo, for their kind assistance with the GPS data used to track the transit vehicles in the transport network of Rome.

## References

1 Ittai Abraham, Daniel Delling, Amos Fiat, Andrew V. Goldberg, and Renato Fonseca F. Werneck. VC-dimension and shortest path algorithms. In Luca Aceto, Monika Henzinger, and Jiri Sgall, editors, *Proc. of the 38th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 690–699. Springer Berlin Heidelberg, 2011.

2 Ittai Abraham, Amos Fiat, Andrew V. Goldberg, and Renato Fonseca F. Werneck. Highway dimension, shortest paths, and provably efficient algorithms. In Moses Charikar, editor, *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 782–793. SIAM, 2010.

3 A. Agresti. *Analysis of Ordinal Categorical Data.* Probability and Statistics. Wiley, 2010.

4 Hannah Bast. Car or public transport—two worlds. In Susanne Albers, Helmut Alt, and Stefan Näher, editors, *Proceedings of the 8th Symposium on Experimental Algorithms (SEA)*, volume 5760 of *Lecture Notes in Computer Science*, pages 355–367. Springer Berlin Heidelberg, 2009.

5 Hannah Bast, Erik Carlsson, Arno Eigenwillig, Robert Geisberger, Chris Harrelson, Veselin Raychev, and Fabien Viger. Fast routing in very large public transportation networks using transfer patterns. In Mark de Berg and Ulrich Meyer, editors, *Proceedings of the 18th annual European conference on Algorithms (ESA): Part I*, pages 290–301. Springer Berlin Heidelberg, 2010.

6 Holger Bast, Stefan Funke, Domagoj Matijevic, Peter Sanders, and Dominik Schultes. In transit to constant time shortest-path queries in road networks. In *Proceedings of the 9th Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 2007.

7 Annabell Berger, Daniel Delling, Andreas Gebhardt, and Matthias Müller-Hannemann. Accelerating time-dependent multi-criteria timetable information is harder than expected. In Jens Clausen and Gabriele Di Stefano, editors, *Proceedings of the 9th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS)*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Germany, 2009.

8 Annabell Berger, Andreas Gebhardt, Matthias Müller-Hannemann, and Martin Ostrowski. Stochastic delay prediction in large train network. In Alberto Caprara and Spyros C. Kontogiannis, editors, *Proceedings of the 9th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS)*, pages 100 – 111. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Germany, 2011.

9 Daniel Delling, Bastian Katz, and Thomas Pajor. Parallel computation of best connections in public transportation networks. *ACM Journal on Experimental Algorithmics*, 17:(4.4), October 2012.

10 Daniel Delling, Thomas Pajor, and Renato Fonseca Werneck. Round-based public transit routing. In David A. Bader and Petra Mutzel, editors, *Proceedings of the 14th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 130–140. SIAM, 2012.

11 Daniel Delling, Peter Sanders, Dominik Schultes, and Dorothea Wagner. Engineering route planning algorithms. In Jürgen Lerner, Dorothea Wagner, and Katharina A. Zweig, editors, *Algorithmics of Large and Complex Networks*, pages 117–139. Springer Berlin Heidelberg, 2009.

12 Julian Dibbelt, Thomas Pajor, Ben Strasser, and Dorothea Wagner. Intriguingly simple and fast transit routing. In Vincenzo Bonifaci, Camil Demetrescu, and Alberto Marchetti-Spaccamela, editors, *Proceedings of the 12th Symposium on Experimental Algorithms (SEA)*, volume 7933 of *Lecture Notes in Computer Science*, pages 43–54. Springer Berlin Heidelberg, 2013.

13 Yann Disser, Matthias Müller–Hannemann, and Mathias Schnee. Multi-criteria shortest paths in time-dependent train networks. In Catherine C. McGeoch, editor, *Proceedings of the 7th Workshop on Experimental Algorithms (WEA)*, pages 347–361. Springer Berlin Heidelberg, 2008.

14 Robert Geisberger. Contraction of timetable networks with realistic transfers. In Paola Festa, editor, *Proceedings of the 9th Symposium on Experimental Algorithms (SEA)*, volume 6049 of *Lecture Notes in Computer Science*, pages 71–82. Springer Berlin Heidelberg, 2010.

15 Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In Catherine C. McGeoch, editor, *Proceedings of the 7th Workshop on Experimental Algorithms (WEA)*, pages 319–333. Springer Berlin Heidelberg, 2008.

16 M. Goerigk, M. Knoth, M. Müller–Hannemann, M. Schmidt, and A. Schöbel. The price of robustness in timetable information. In Alberto Caprara and Spyros C. Kontogiannis, editors, *Proceedings of the 9th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS)*, pages 76 – 87. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Germany, 2011.

17 Andrew V. Goldberg. The hub labeling algorithm. In Vincenzo Bonifaci, Camil Demetrescu, and Alberto Marchetti-Spaccamela, editors, *Proceedings of the 12th Symposium on Experimental Algorithms (SEA)*, page 4. Springer Berlin Heidelberg, 2013.

18 Andrew V. Goldberg, Haim Kaplan, and Renato F. Werneck. Reach for A*: Efficient point-to-point shortest path algorithms. In Rajeev Raman and Matthias F. Stallmann, editors, *Proceedings of the 8th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 129–143. SIAM, 2006.

19 Ronald J. Gutman. Reach-based routing: A new approach to shortest path algorithms optimized for road networks. In Lars Arge, Giuseppe F. Italiano, and Robert Sedgewick,

editors, *Proceedings of the Sixth Workshop on Algorithm Engineering and Experiments and the First Workshop on Analytic Algorithmics and Combinatorics*, pages 100–111. SIAM, 2004.

20 Timothy Hunter, Ryan Herring, Pieter Abbeel, and Alexandre Bayen. Path and travel time inference from GPS probe vehicle data. In Daphne Koller, Yoshua Bengio, Léon Bottou, and Aron Culotta, editors, *Advances in Neural Information Processing Systems 21*. Nips Foundation, 2009.

21 Erik Jenelius and Haris N. Koutsopoulos. Travel time estimation for urban road networks using low frequency probe vehicle data. *Transportation Research Part B: Methodological*, 53(0):64 – 81, 2013.

22 M. Kendall. A new measure of rank correlation. *Biometrika*, 30(1–2):81–89, 1938.

23 Matthias Müller-Hannemann and Mathias Schnee. Efficient timetable information in the presence of delays. In Christos D. Zaroliagis Ravindra K. Ahuja, Rolf H. Möhring, editor, *Robust and Online Large-Scale Optimization*, pages 249–272. Springer Berlin Heidelberg, 2009.

24 Matthias Müller-Hannemann, Frank Schulz, Dorothea Wagner, and Christos D. Zaroliagis. Timetable information: Models and algorithms. In Frank Geraets, Leo Kroon, Anita Schoebel, Dorothea Wagner, and Christos D. Zaroliagis, editors, *Algorithmic Methods for Railway Optimization*, pages 67–90. Springer Berlin Heidelberg, 2007.

25 Evangelia Pyrga, Frank Schulz, Dorothea Wagner, and Christos D. Zaroliagis. Efficient models for timetable information in public transportation systems. *ACM Journal on Experimental Algorithmics*, 12:(2.4), 2008.

26 Peter Sanders and Dominik Schultes. Highway hierarchies hasten exact shortest path queries. In Gerth Stølting Brodal and Stefano Leonardi, editors, *Proceedings of the 13th annual European conference on Algorithms (ESA)*, pages 568–579. Springer Berlin Heidelberg, 2005.

27 Jing Yuan, Yu Zheng, Xing Xie, and Guangzhong Sun. Driving with knowledge from the physical world. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '11, pages 316–324, New York, NY, USA, 2011. ACM.

28 Agenzia Roma servizi per la Mobilità. Muoversi a Roma. `http://muovi.roma.it/`, 2013. [Online; accessed June-2013].

# Robust Routing in Urban Public Transportation: How to find reliable journeys based on past observations [*]

**Kateřina Böhmová, Matúš Mihalák, Tobias Pröger, Rastislav Šrámek, and Peter Widmayer**

**Institute of Theoretical Computer Science, ETH Zurich, Switzerland**
`{kboehmov,mmihalak,tproeger,rsramek,widmayer}@inf.ethz.ch`

## Abstract

We study the problem of robust routing in urban public transportation networks. In order to propose solutions that are robust for typical delays, we assume that we have past observations of real traffic situations available. In particular, we assume that we have "daily records" containing the observed travel times in the whole network for a few past days. We introduce a new concept to express a solution that is feasible in any record of a given public transportation network. We adapt the method of Buhmann et al. [4] for optimization under uncertainty, and develop algorithms that allow its application for finding a robust journey from a given source to a given destination. The performance of the algorithms and the quality of the predicted journey are evaluated in a preliminary experimental study. We furthermore introduce a measure of reliability of a given journey, and develop algorithms for its computation. The robust routing concepts presented in this work are suited specially for public transportation networks of large cities that lack clear hierarchical structure and contain services that run with high frequencies.

**1998 ACM Subject Classification** F.2.2 Nonnumerical Algorithms and Problems, G.2.2 Graph Theory (Graph algorithms, Network problems), I.2.6 Learning

**Keywords and phrases** Algorithms, Optimization, Robustness, Route planning, Public transportation

**Digital Object Identifier** 10.4230/OASIcs.ATMOS.2013.27

## 1 Introduction

We study the problem of routing in urban public transportation networks, such as tram and bus networks in large cities, focusing on the omnipresent uncertain situations when (typical) delays occur. In particular, we search for robust routes that allow reliable yet quick passenger transportation. We think of a "dense" tram network in a large city containing many tram lines, where each tram line is a sequence of stops that is served repeatedly during the day, and where there are several options to get from one location to another. Such a network usually does not contain clear hierarchical structure (as opposed to train networks), and each line is served with high frequency. Given two tram stops $a$ and $b$ together with a latest arrival time $t_A$, our goal is to provide a simple yet robust description of how to travel in the

[*] This work has been partially supported by the Swiss National Science Foundation (SNF) under the grant number 200021 138117/1, and by the EU FP7/2007-2013 (DG CONNECT.H5-Smart Cities and Sustainability), under grant agreement no. 288094 (project eCOMPASS). Kateřina Böhmová is a recipient of the Google Europe Fellowship in Optimization Algorithms, and this research is supported in part by this Google Fellowship.

given network from $a$ to $b$ in order to arrive on time $t_A$ even in the presence of typical delays. We base our robustness concepts on past traffic data in a form of recorded timetables – the actually observed travel times of all lines in the course of several past days. If no delays occur, such a recorded timetable corresponds to the scheduled timetable for that day.

The standard approach to describe a travel plan from $a$ to $b$ in a given tram network is to specify, according to a scheduled timetable, the concrete sequence of vehicles together with transfer stops and departure/arrival times for each transfer stop. Such a travel plan may look like this: Take the tram 6 at 12:33 from stop $a$ and leave it at 12:47 at transfer stop $s$; then take the tram 10 at 12:51 from $s$ and leave it at 12:58 at $b$. However, such a travel plan may become infeasible on a concrete day due to delays: Imagine a situation where the tram 6 left $a$ at 12:33, but arrived to $s$ only at 12:53, and the tram 10 leaving $s$ at 12:51 was on time. Then, the described travel plan would bring the passenger to stop $s$ but it does not specify how to proceed further in order to arrive to $b$.

We observe that the standard solution concepts (such as paths in a time-expanded graph) are not suitable for our setting. We introduce a new concept to express a solution, which we call a *journey*, that is feasible in any recorded timetable of a given transportation network assuming the timetable to be periodic. A journey specifies an initial time $t_D$ and then only a sequence of transportation lines $\langle l_1(\text{tram}), l_2(\text{bus}), \ldots, l_k(\text{tram})\rangle$ together with transfer stops $\langle s_1, \ldots, s_{k-1}\rangle$. This travel plan suggests to start waiting at $a$ at time $t_D$, take the first tram of line $l_1$ that comes and travel to stop $s_1$, then change to the first coming bus of line $l_2$, etc. Since we assume that the frequency of vehicles serving each line is high, such a travel plan is not only feasible in our setting but also reasonable, and provides the passenger with all the necessary information. We provide algorithms to efficiently compute these journeys.

Equipped with the introduced solution concept of a journey, we can easily adapt the method of Buhmann et al. [4] for optimization under uncertainty, and apply it to identify robust travel plans. A key ingredient of the method is the ability to count the number of (possibly exponentially many) "good" solutions. Our solution concept allows us to develop efficient algorithms to compute the number of all journeys from $a$ to $b$ that depart after the time $t_D$ and arrive before the time $t_A$.

Finally, we suggest an alternative simple measure for reliability of a given journey, expressed simply as the fraction of recorded timetables where the journey was successful and allowed to arrive at the destination on time. We provide efficient algorithms for computation of this measure.

## 2 Related Work

The problem of finding a fastest journey (according to the planned timetable) using public transportation has been extensively studied in the literature. Common approaches model the transportation network as a graph and compute a shortest path in this graph (see [12] for a survey). Various improvements have been developed, and experimental studies suggest that these can also be used in practice (see, e.g., [2, 5, 14]). Recent approaches avoid the construction of a graph and process the timetable directly [6, 7]. For example, Delling et al. [6] describe an approach which is centered around transportation lines (such as train or bus lines) and which can be used to find all pareto-optimal journeys when the arrival time and the number of stops are considered as criteria. Bast et al. [1] observe that for two given stops, we can find and encode each sequence of intermediate transfer stations (i.e., stations where we change from one line to another) that can lead to an optimal route. The set of these sequences of transfers is called *transfer pattern*. These patterns can be precomputed,

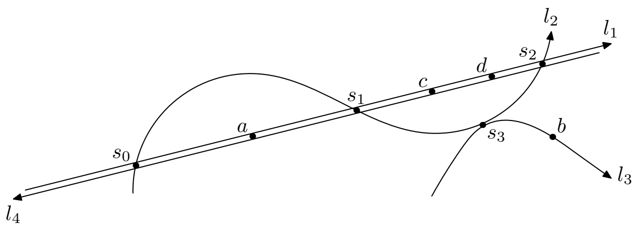


**Figure 1** The line $l_1$ is a sequence of stops $\langle \dots, s_0, a, s_1, c, d, s_2, \dots \rangle$. The line $l_4 = \langle \dots, s_2, d, c, s_1, b, s_0, \dots \rangle$ that goes in the opposite direction to $l_1$ is considered to be a different line. In this example, both $a \lhd l_1$ and $a \lhd l_4$ hold, but $a \lhd l_2$ does not. Similarly, $s_1 \lhd s_2 \lhd l_1$ holds, but $s_1 \lhd s_2 \lhd l_4$ does not. The set $l_1 \cap l_2$ of all stops common to $l_1$ and $l_2$ is $\{s_0, s_1, s_2\}$. Moreover, when travelling from $a$ to $b$ using a route $\langle l_1, l_2, l_3 \rangle$, this network is an example where not every stop in $l_1 \cap l_2$ is suitable for changing from $l_1$ to $l_2$: We cannot choose $s_0$ as transfer stop since it is served before $a$. If $s_2$ was chosen, then $l_3$ can never be reached without travelling back. Thus, the only valid stop to change the line is $s_1$.

leading to very fast query times. These approaches are similar to our approach in the sense that they try to explicitly exploit the problem structure (e.g., by considering lines) instead of implicitly modelling all properties into a graph.

For computing *robust* journeys in public transportation, stochastic networks have been studied [3, 9, 13], where the delays between successive edges are random variables. Dibbelt et al. [7] study the case when stochastic delays on the vehicles are given. In a situation when timetables are fixed, Disser et al. [8] used a generalization of Dijkstra's algorithm to compute pareto-optimal multi-criteria journeys. They define the reliability of a journey as a function depending on the minimal time to change between two subsequent trains, and use it as an additional criterion. Müller-Hannemann and Schnee [11] introduced the concept of a *dependency graph* for a prediction of secondary delays caused by some current primary delays, which are given as input. They also show how to compute a journey that is optimal with respect to the predicted delays. Goerigk et al. [10] consider a given set of delay *scenarios* for every event, and adapt *strict robustness* to it, i.e. they aim to compute a journey that arrives on time for every scenario. Furthermore, the concept of *light robustness* is introduced, which aims to compute a journey that maximizes the number of scenarios in which the travel time of this journey lies at most a fixed time above the optimum. Strict robustness requires a feasible solution for every realization of delays for every event. This is quite conservative, as in reality not every combination of event delays appears. Our approach tries to avoid this by learning from the typical delay scenarios as recorded for each individual day.

## 3 Modeling issues

### 3.1 Model

**Stops and lines** Let $\mathcal{S}$ be a set of stops, and $\mathcal{L} \subset \bigcup_{i=2}^{|\mathcal{S}|} \mathcal{S}^i$ be a set of lines (e.g., bus lines, tram lines or lines of other means of transportation). The following basic definitions are illustrated in Figure 1. Every line $l \in \mathcal{L}$ is a sequence of $S(l)$ stops $\langle s_1^{(l)}, \dots, s_{S(l)}^{(l)} \rangle$, where, for every $i \in \{1, \dots, S(l) - 1\}$, the stop $s_i^{(l)}$ is served directly before $s_{i+1}^{(l)}$ by the line $l$. We explicitly distinguish two lines that serve the same stops but have opposite directions (these may be operated under the same identifier in reality). For a stop $s \in \mathcal{S}$ and a line $l \in \mathcal{L}$, we write $s \lhd l$ if $s$ is a stop on the line $l$, i.e. if there exists an index $i \in \{1, \dots, S(l)\}$ such that $s = s_i^{(l)}$. Furthermore, for two stops $s_1, s_2 \in \mathcal{S}$ and a line $l \in \mathcal{L}$ we write $s_1 \lhd s_2 \lhd l$ if both $s_1$ and $s_2$ are stops on $l$ and $s_1$ is served before $s_2$, i.e. if there exist indices $i, j \in \mathbb{N}$, $1 \leq i \leq S(l) - 1$, $i + 1 \leq j \leq S(l)$ such that $s_1 = s_i^{(l)}$ and $s_2 = s_j^{(l)}$. For two lines $l_1, l_2 \in \mathcal{L}$, we define $l_1 \cap l_2$ to be the set of all stops $s \in \mathcal{S}$ that are served both by $l_1$ and $l_2$.

**Trips and timetables** While the only information associated with a line itself are its consecutive stops, it usually is operated multiple times per day. Each of these concrete realizations that departs at a given time of the day is called a *trip*. With every trip $\tau$ we associate a line $L(\tau) \in \mathcal{L}$. By $L^{-1}(l)$ we denote the set of *all* trips associated with a line $l \in \mathcal{L}$. For a trip $\tau$ and a stop $s \in \mathcal{S}$, let $A(\tau, s)$ be the arrival time of $\tau$ at stop $s$, if $s \lhd L(\tau)$. Analogously, let $D(\tau, s)$ be the departure time of $\tau$ at $s$. In the following, we assume time to be modelled by integers. For a given trip $\tau$, we require $A(\tau, s) \leq D(\tau, s)$ for every stop $s \in L(\tau)$. Furthermore we require $D(\tau, s_1) \leq A(\tau, s_2)$ for every two stops $s_1, s_2 \in \mathcal{S}$ with $s_1 \lhd s_2 \lhd L(\tau)$. A set of trips is called a *timetable*. We distinguish between

1. the *planned* timetable $T$. We assume it to be periodic, i.e., every line realized by some trip $\tau$ will be realized by a later trip $\tau'$ again (probably not on the same day).
2. *recorded* timetables $T_i$ that describe how various lines were operated during a given time period (i.e., on a concrete day or during a concrete week). These recorded timetables are concrete executions of the planned timetable.

In the following, *timetable* refers both to the planned as well as to a recorded timetable.

**Goal** In the following, let $a, b \in \mathcal{S}$ be two stops, $m \in \mathbb{N}_0$ be the maximal allowed number of line changes, and $t_A \in \mathbb{N}$ be the latest arrival time. A *journey* consists of a departure time $t_D$, a sequence of lines $\langle l_1, \ldots, l_k \rangle$, $k \leq m + 1$ and a sequence of transfer stops $\langle s_{\text{CH}}^{(1)}, \ldots, s_{\text{CH}}^{(k-1)} \rangle$. The intuitive interpretation of such a journey is to start at stop $a$ at time $t_D$, take the first line $l_1$ (more precisely, the first available trip of the line $l_1$), and for every $i \in \{1, \ldots, k-1\}$, leave $l_i$ at stop $s_{\text{CH}}^{(i)}$ and take the next arriving line $l_{i+1}$ immediately. Our goal is to compute a recommendation to the user in form of one or more (robust) journeys from $a$ to $b$ that will likely arrive on time (i.e., before time $t_A$) on a day for which the concrete travel times are not known yet. We formalize the notion of robustness later. We note that for the convenience of the user, one should handle two different lines $l_1$ and $l_2$ operating between two stops $s_1$ and $s_2$ as one (virtual) line, and provide recommendations of the form "in $s_1$, take the first line $l_1$ or $l_2$ to $s_2$, etc.". For the sake of simplicity we do not pursue this generalization further, but will consider this in the future.

**Routes** Let $k \in \{1, \ldots, m + 1\}$ be an integer. A sequence of lines $r = \langle l_1, \ldots, l_k \rangle \in \mathcal{L}^k$ is called a *feasible route from $a$ to $b$* if there exist $k + 1$ stops $s_0 := a, s_1, \ldots, s_{k-1}, s_k := b$ such that $s_{i-1} \lhd s_i \lhd l_i$ for every $i \in \{1, \ldots, k\}$, i.e., if both $s_{i-1}$ and $s_i$ are stops on line $l_i$, and $s_{i-1}$ is served before $s_i$ on line $l_i$. Notice that on a feasible route $r \in \mathcal{L}^k$ we need to change the line at $k - 1$ transfer stops. Let

$$\mathcal{R}_{ab}^m = \{r \in \mathcal{L} \cup \mathcal{L}^2 \cup \cdots \cup \mathcal{L}^{m+1} \mid r \text{ is a feasible route from } a \text{ to } b\} \tag{1}$$

be the set of all feasible routes from $a$ to $b$ using at most $m$ transfer stops. If $a$, $b$ and $m$ are clear from the context, for simplicity we just write $\mathcal{R}$ instead of $\mathcal{R}_{ab}^m$. Notice that by definition, a line $l$ may occur multiple times in a route. This is reasonable because there might be two transfer stops $s, s'$ on $l$ and one or more intermediate lines that travel faster from $s$ to $s'$ than $l$ does. Additionally, notice that a route does not contain *any* time information.

## 3.2 Computation of Feasible Routes

**Input data** In this section we describe an algorithm that, given a set of stops $\mathcal{S}$ and a set of lines $\mathcal{L}$, finds the set $\mathcal{R}$ of all feasible routes that allow to travel from a given initial stop $a$ to a given destination stop $b$ using at most $m$ transfer stops (also called transfers). Notice that to compute $\mathcal{R}$ we only need the network structure, no particular timetable is necessary.

**Preprocessing the input** We preprocess the input data and construct data structures to allow efficient queries of the following types:

1. $Q(l, s)$: *Compute the position of $s$ on $l$.* Given a line $l = \langle s_1, \dots, s_{|S(l)|} \rangle$, $Q(l, s)$ returns $j$ if $s$ is the $j$-th stop on $l$, i.e., if $s = s_j$, or 0 if $s$ is not served by $l$.
2. $Q(l, s_i, s_j)$: *Determine whether $s_i$ is served before $s_j$ on $l$.* Given a line $l$ and two stops $s_i, s_j$, $Q(l, s_i, s_j)$ returns `TRUE` iff $s_i, s_j \lhd l$ and $s_i \lhd s_j \lhd l$.
3. $Q(l_i, l_j)$: *Determine $l_i \cap l_j$ (i.e., the stops shared by $l_i$ and $l_j$) in a compact, ordered format.* Given two lines $l_i$, and $l_j$, $Q(l_i, l_j)$ returns the set $l_i \cap l_j$ of stops shared by these lines. We encode $l_i \cap l_j$ into an ordered set $I_{ij}$ of pairs of stops with respect to $l_i$ in such a way that $(s_q, s_r) \in I_{ij}$ indicates that $l_i$ and $l_j$ share the stops $s_q$, $s_r$, and all the stops in between on the line $l_i$ (independent of their order on $l_j$). Thus, $Q(l_i, l_j)$ outputs the described sorted set $I_{ij}$ of pairs of stops. The motivation to compress $l_i \cap l_j$ into $I_{ij}$ is that, in practice, there may be many stops shared by $l_i$ and $l_j$, but only a small number of contiguous intervals of such stops. Notice that $Q(l_i, l_j)$ doesn't need to be equal to $Q(l_j, l_i)$, nor the sequence in reverse order; an example is given in Figure 2.

Notice that these queries can be answered in expected constant time if implemented using suitable arrays or hashing tables.

**Graph of line incidences** The function $Q(l_i, l_j)$ induces the following directed graph $G$. The set $V$ of vertices of $G$ corresponds to the set of lines $\mathcal{L}$. There is an edge from a vertex (line) $l_i$ to a vertex $l_j$ if and only if $Q(l_i, l_j) \neq \emptyset$. Then, $Q(l_i, l_j)$ is represented as a tag of the edge $(l_i, l_j)$. We construct and represent the graph $G$ as adjacency lists.

**Preliminary observations** Given two stops $a$ and $b$, and a number $m$, we want to find all routes $\mathcal{R}$ that allow to travel from $a$ to $b$ using at most $m$ transfers in the given public transportation network described by a set of stops $\mathcal{S}$ and a set of lines $\mathcal{L}$. Notice that each such route $r = \langle l_1, \dots, l_k \rangle \in \mathcal{L}^k$ with $0 < k \leq m + 1$ has the following properties.

1. Both $Q(l_1, a)$ and $Q(l_k, b)$ are nonzero (i.e., $a \lhd l_1$, and $b \lhd l_k$).
2. The vertices $l_1, \dots, l_k$ form a path in $G$ (i.e., $l_i \cap l_{i+1} \neq \emptyset$ for every $i = 1, \dots, k-1$).
3. There exists a sequence of stops $a = s_0, s_1, \dots, s_{k-1}, s_k = b$ such that $Q(l_i, s_{i-1}, s_i)$ is `TRUE` (i.e., $s_{i-1} \lhd s_i \lhd l_i$) for every $i = 1, \dots, k$.

These observations lead to the following algorithm to find the set of routes $\mathcal{R}$.

**All routes algorithm** For the stop $a$, determine the set $\mathcal{L}_a$ of all lines passing through $a$. Then explore the graph $G$ from the set $\mathcal{L}_a$ of vertices in the following fashion. For each vertex $l_1 \in \mathcal{L}_a$, perform a depth-first search in $G$ up to the depth $m$, but do not stop when finding a vertex that has already been found earlier. In each step, try to extend a partial path $\langle l_1, \dots, l_j \rangle$ to a neighbor $l'_j$ of $l_j$ in $G$. Keep track of the *current transfer stop* $s_q$. This is a stop on the currently considered line $l_j$ such that $s_q$ is the stop with the smallest position on $l_j$ at which it is possible to transfer from $l_{j-1}$ to $l_j$, considering the partial path from $l_1$ to $l_{j-1}$. In other words, $s_q$ is the stop on the considered route where the line $l_j$ can be boarded. Each step of the algorithm is characterized by a *search state*: a partial path $P = \langle l_1, \dots, l_j \rangle$, and a current transfer stop $s_q$ that allowed the transfer to line $l_j$. The initial search state consists of the partial path $P = \langle l_1 \rangle$ and the current transfer stop $a$. More specifically, to

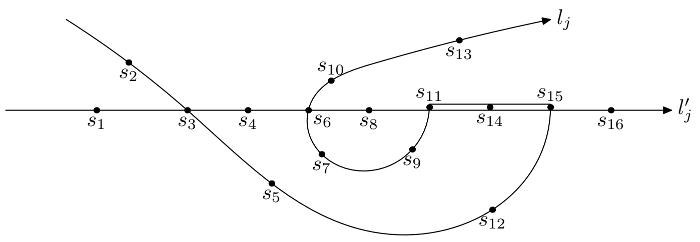


**Figure 2** Lines $l_j$ and $l'_j$ have common stops $s_3$, $s_6$, $s_{11}$, $s_{14}$, and $s_{15}$. The ordered set $I_{jj'} = Q(l_j, l'_j)$ consists of the pairs $\{(s_3, s_3), (s_{15}, s_{11}), (s_6, s_6)\}$. Thus, the last stop in the last interval of $I_{jj'}$ is the stop $s_6$. On the other hand, the ordered set $I_{j'j} = Q(l'_j, l_j)$ consists of the pairs $\{(s_3, s_3), (s_6, s_6), (s_{11}, s_{15})\}$. Now, imagine that the current transfer stop $s_q$ for a partial path $P = \langle l_1, \ldots, l_j \rangle$ is $s_2$, then the stop $s_3$ is the current transfer stop $s'_q$ for a partial path $P' = \langle l_1, \ldots, l_j, l'_j \rangle$. However, observe that if $s_q$ is $s_{12}$, then $s'_q$ needs to be $s_6$.

process a search state with the partial path $P = \langle l_1, \ldots, l_j \rangle$, and the current transfer stop $s_q$, perform the following tasks:

1. Check whether the line corresponding to the vertex $l_j$ contains the stop $b$ and whether $s_q$ is before $b$ on $l_j$. If this is the case (i.e., the query $Q(l_j, s_q, b)$ returns `TRUE`), then the partial path $P$ corresponds to a feasible route and is output as one of the solutions in $\mathcal{R}$.
2. If the partial path $P$ contains at most $m-1$ edges (thus the corresponding route has at most $m-1$ transfers, and can be extended), then for each neighbor $l'_j$ of $l_j$ check whether extending $P$ by $l'_j$ is possible (and if so, update the current transfer stop) as follows. Let $I_{jj'} = Q(l_j, l'_j)$ be the set of pairs of stops sorted as described in the previous section. Recall that each pair $(s_u, s_v) \in I_{jj'}$ encodes an interval of one or several consecutive stops on $l_j$ that are also stops on the line $l'_j$. Let $s_z$ be the last stop in the last interval of $I_{jj'}$. Similarly, let $I_{j'j} = Q(l'_j, l_j)$. If $Q(l_j, s_q, s_z)$ is `TRUE`, then $s_q \lhd s_z \lhd l_j$, and the path $P$ can be extended to $l'_j$.
   a. We determine the current transfer stop $s'_q$ for $l'_j$ by considering the pairs/intervals of $I_{j'j}$ in ascending order and deciding whether the position of $s_q$ on the line $l_j$ is before one of the endpoints of the currently considered interval. We refer to Figure 2 for a nontrivial case of computing of the current transfer stop.
   b. Perform the depth search with the search state consisting of the partial path $P' = \langle l_1, \ldots, l_j, l'_j \rangle$ and the current transfer stop $s'_q$.

   Otherwise, if $Q(l_j, s_q, s_z)$ is `FALSE`, it is not possible to extend $P$ to $l'_j$.

The theoretical running time of the algorithm is $\mathcal{O}(\Delta^m)$, where $\Delta$ is the maximum degree of $G$. However, we believe that in practice the actual running time will rather linearly correspond to the size of the output $\mathcal{O}(m|\mathcal{R}|)$. On real-world data, the algorithm performs reasonably fast (see section 6 for details).

## 3.3 Computing the earliest arriving journey

**Recursive computation** As previously stated, let $a \in \mathcal{S}$ be the initial stop, $b \in \mathcal{S}$ be the destination stop, $\varepsilon(s, l, l')$ be the minimum time to change from line $l$ to line $l'$ at station $s$, and $t_A \in \mathbb{N}$ be the latest arrival time. In the previous section we showed how the set $\mathcal{R}$ of all feasible routes from $a$ to $b$ can be computed. However, instead of presenting just a route $r \in \mathcal{R}$ to the user, our final goal is to compute a departure time $t_0$ and a *journey* that arrives at $b$ before time $t_A$. For the following considerations, we assume the underlying timetable (either the planned or a recorded timetable) to be fixed. Given $a, b \in \mathcal{S}$, an initial departure time $t_0 \in \mathbb{N}$, and a route $r = \langle l_1, \ldots, l_k \rangle \in \mathcal{R}$, a journey along $r$ that arrives as early as possible can be computed as follows. We start at $a$ at time $t_0$ and take the first line $l_1$ that arrives. Then we compute an appropriate transfer stop $s \in l_1 \cap l_2$ (that is served both by $l_1$ as well as by $l_2$) and the arrival time $t_1$ at $s$, leave $l_1$ there and compute recursively the earliest arrival time when departing from $s$ at time at least $t_1 + \varepsilon(s, l_1, l_2)$, following the route $\langle l_2, \ldots, l_k \rangle$. Notice that the selection of an appropriate transfer stop $s$ is the only non-trivial part due to mainly two reasons:

1. The lines $l_1$ and $l_2$ may operate with different speeds (e.g., because $l_1$ is a fast tram while $l_2$ is a slow bus), or $l_1$ and $l_2$ separate at a stop $s_1$ and join later again at a stop $s_2$ but the overall travel times of $l_1$ and $l_2$ differ between $s_1$ and $s_2$. Depending on the situation, it may be better to leave $l_1$ as soon or as late as possible, or anywhere inbetween.
2. The lines $l_1$ and $l_2$ may separate at a stop $s_1$ and join later again at a stop $s_2$. If all transfer stops in $l_2 \cap l_3$ are served by $l_2$ before $s_2$, then leaving $l_1$ at $s_2$ is not an option since $l_3$ is not reachable anymore. See Figure 1 for a visualization.

The idea now is to find the earliest trip of line $l_1$ that departs from $a$ at time $t_0$ or later, iterate over all stops $s \in l_1 \cap l_2$, and compute recursively the earliest arrival time when continuing the journey from $s$ having a changing time of at least $\varepsilon(s, l_1, l_2)$. Finally, we return the smallest arrival time that was found in one of the recursive calls.

**Issues and improvement of the recursive algorithm** An issue with this naïve implementation is the running time, which might be exponential in $k$ in the worst-case (if $|l_i \cap l_{i+1}| > 1$ for $\Omega(k)$ many $i \in \{1, \ldots, k-1\}$). Let $\tau$ and $\tau'$ be two trips with $L(\tau) = L(\tau')$. If $\tau$ leaves before $\tau'$ at some stop $s$, we assume that it will never arrive later than $\tau'$ at any subsequent stop $s'$, $s \lhd s' \lhd L(\tau)$, i.e. consecutive trips of the same line do not overtake. For a line $l \in \mathcal{L}$ and a set of trips $T_l \subseteq L^{-1}(l)$, it follows that taking the earliest trip in $T_l$ never results in a later arrival at $b$ than taking any other trip from $T_l$. Furthermore, a trip $\tau \in T_l$ is operated earlier than a trip $\tau' \in T_l$ iff $A(\tau, s) < A(\tau', s)$ for *any* stop $s \lhd l$.

Thus, we can iterate over some appropriate stops in $l_1 \cap l_2$ to find the earliest reachable trip associated with $l_2$. We just need to ignore those stops where changing to $l_3$ is no longer possible (see Figure 1 for an example).

**Computing appropriate transfer stops** The problem to find these appropriate stops can be solved by first sorting $l_1 \cap l_2 = \{s_1, \ldots, s_n\}$ such that $s_j \lhd s_{j+1} \lhd l_1$ for every $j \in \{1, \ldots, n-1\}$. Obviously, all stops that appear before $a$ on line $l_1$ cannot be used for changing to $l_2$. This problem can easily be solved by considering only those stops $s_j$ where $a \lhd s_j \lhd l_1$. Unfortunately, the last $m \geq 0$ stops $s_{n-m+1}, \ldots, s_n$ might also not be suitable for changing to $l_2$ because they may prevent us later to change to some line $l_j$ (e.g., if *all* stops of $l_2 \cap l_3$ are served before $s_{n-g+1}, \ldots, s_n$ on $l_2$, then changing to $l_3$ is no longer possible). We solve this problem by precomputing (the index of) the last stop $s_j$ where all later lines are still reachable. This can be done backwards: we start at $b$, order the elements of $l_k \cap l_{k-1}$ as they appear on line $l_k$, and find the last stop that is served before $b$ on $l_k$. We recursively continue with $l_1, \ldots, l_{k-1}$ and use the stop previously computed as the stop that still needs to be reachable.

**Iterative algorithm** The improved algorithm first iterates over $i \in \{1, \ldots, k-1\}$, and uses the aforementioned algorithm to precompute the index last$[i]$ of the last stop where changing from $l_i$ to $l_{i+1}$ is still possible (with respect to the route $\langle l_1, \ldots, l_k \rangle$). After that, for every $i \in \{1, \ldots, k-1\}$, we iterate over the appropriate transfer stops $s \in l_i \cap l_{i+1}$ where changing to $l_{i+1}$ is possible, and find among those the stop $s_{\text{CH}}^{(i)}$ where the earliest trip $\tau_{i+1}$ associated with line $l_{i+1}$ departs. Finally we obtain a sequence of trips $\tau_1, \ldots, \tau_k$ along with transfer stops $s_{\text{CH}}^{(0)} := a, s_{\text{CH}}^{(1)}, \ldots, s_{\text{CH}}^{(k)}$ to change lines. Since we gradually compute the earliest trips $\tau_i$ for each of the lines $l_i$, the earliest time to arrive at $b$ is simply $A(\tau_k, b)$.

```
EarliestArrival($a, b, t_0, \langle l_1, \dots, l_k \rangle$)
1  last[$k$] $\leftarrow b$
2  for $i \leftarrow k, \dots, 2$ do
3      Order the elements of $l_i \cap l_{i-1} = \{s_1, \dots, s_n\}$ s.t. $s_j \lhd s_{j+1} \lhd l_{i-1}\ \forall j \in \{1, \dots, n-1\}$.
4      last[$i-1$] $\leftarrow \max\{j \in \{1, \dots, n\} \mid s_j \lhd \text{last}[i] \lhd l_i\}$
5  $\tau_1 \leftarrow \arg\min_{\tau \in L^{-1}(l_1)}\{D(\tau, a) \mid D(\tau, a) \geq t_0\}$;  $s^{(0)}_{\text{CH}} \leftarrow a$
6  for $i \leftarrow 1, \dots, k-1$ do
7      Order the elements of $l_i \cap l_{i+1} = \{s_1, \dots, s_n\}$ s.t. $s_j \lhd s_{j+1} \lhd l_i\ \forall j \in \{1, \dots, n-1\}$.
8      $\tau_{i+1} \leftarrow$ null;  $s^{(i)}_{\text{CH}} \leftarrow$ null;  $A^{(i+1)}_{s_n} \leftarrow \infty$
9      for $j \leftarrow 1, \dots, \text{last}[i]$ do
10         if $s^{(i-1)}_{\text{CH}} \lhd s_j \lhd l_i$ and $s_j \lhd \text{last}[i+1] \lhd l_{i+1}$ then
11             $\tau' \leftarrow \arg\min_{\tau \in L^{-1}(l_{i+1})}\{D(\tau, s_j) \mid D(\tau, s_j) \geq A(\tau_i, s_j) + \varepsilon(s_j, l_i, l_{i+1})\}$
12             if $A(\tau', s_n) < A^{(i+1)}_{s_n}$ then $\tau_{i+1} \leftarrow \tau'$;  $s^{(i)}_{\text{CH}} \leftarrow s_j$;  $A^{(i+1)}_{s_n} \leftarrow A(\tau', s_n)$
13 return $A(\tau_k, b)$
```

Let $n = \max\{|l_i \cap l_{i+1}|\}$. Given a line $l \in \mathcal{L}$, a station $s \in \mathcal{S}$ and a time $t_0 \in \mathbb{N}$, let $f$ be the time to find the earliest trip $\tau$ with $L(\tau) = l$ und $D(\tau, s) \geq t_0$ (this time depends on the concrete implementation of the timetable). It is easy to see that the running time of the above algorithm is bounded by $\mathcal{O}(kn(\log n + f))$.

## 4 Maximizing the Unexpected Similarity

**Computing the optimum journey for a fixed timetable** Given two stops $a, b \in \mathcal{S}$ and a departure time $t_0 \in \mathbb{N}$, we can already compute the earliest arrival of a journey from $a$ to $b$ starting at time $t_0$. From now on, we aim to compute the latest departure time at $a$ when the latest *arrival* time $t_A$ at $b$ is given. For this purpose we present an algorithm that sweeps backwards in time and uses the previous algorithm Earliest-Arrival. This sweepline algorithm will later be extended to count journeys (instead of computing a single one) and can be used for finding robust journeys, i.e. journeys that are likely to arrive on time.

The sweepline algorithm works as follows. We consider the trips departing at stop $a$ before time $t_A$, sorted in reverse chronological order. Everytime we find a trip $\tau$ of any line departing at some time $t_0$, we check whether there exists a route $r = \langle L(\tau), l_2, \dots, l_k \rangle \in \mathcal{R}$ that starts with the line $L(\tau)$. If yes, then we use the previous algorithm to compute the earliest arrival time at $b$ when we depart at $a$ at time $t_0$ and follow the route $r$. If the time computed is not later than $t_A$, we found the optimal solution and stop the algorithm. Otherwise we continue with the previous trip departing from $a$.

**Finding robust journeys** We will now describe how to compute robust journeys using the approach of Buhmann et al. [4]. We stress up front that this is "learning"-style algorithm and that it, in particular, does not specifically aims at optimizing some "robustness" criterion (such as the fraction of successes in the recorded timetables). Let $a, b \in \mathcal{S}$ be the departure and the target stop of the journey, $t_A$ be the latest arrival time at $b$, and $\mathcal{T}$ be a set of recorded timetables for comparable time periods (e.g., daily recordings for the past Mondays). For a timetable $T \in \mathcal{T}$ and a value $\gamma$, the *approximation set* $A_\gamma(T)$ contains a route $r \in \mathcal{R}$ iff there exists a journey along the route $r$ that starts at $a$ at time $t_A - \gamma$ or later and arrives at $b$ at time $t_A$ or earlier (both times refer to timetable $T$). The major advantage of this definition over classical approximation definitions (such as multiplicative approximation) is

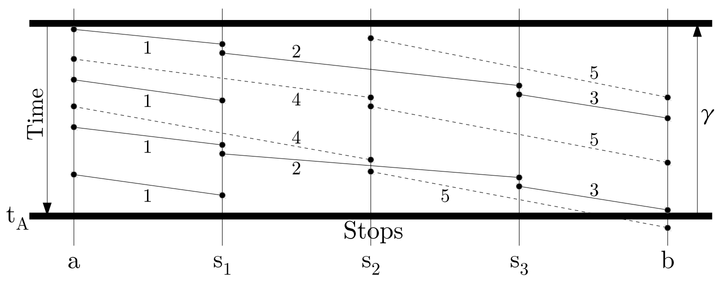


**Figure 3** An example with five lines $\{1,\ldots,5\}$ and two routes $r_1 = \langle 1,2,3\rangle$ (solid) and $r_2 = \langle 4,5\rangle$ (dotted). The $x$-axis illustrates the stops $\{a, s_1, s_2, s_3, b\}$, whereas the $y$-axis the time. If a trip leaves a stop $s_d$ at time $t_d$ and arrives at a stop $s_a$ at time $t_a$, it is indicated by a line segment from $(s_d, t_d)$ to $(s_a, t_a)$. We have $\mu^T_\gamma(r_1) = 3$ and $\mu^T_\gamma(r_2) = 1$.

that we can consider multiple recorded timetables at the same time, and that the parameter $\gamma$ still has a direct interpretation as the time that we depart before $t_A$. Especially, if we consider approximation sets $A_\gamma(T_1), \ldots, A_\gamma(T_k)$ for $T_1, \ldots, T_k \in \mathcal{T}$, every set contains only routes that appear in the same time period and are therefore comparable among different approximation sets.

To identify *robust* routes when only two timetables $T_1, T_2 \in \mathcal{T}$ are given, we consider $A_\gamma(T_1) \cap A_\gamma(T_2)$: the only chance to find a route that is likely to be good in the future is a route that was good in the past for *both* recorded timetables. The parameter $\gamma$ determines the size of the intersection: if $\gamma$ is too small, the intersection will be empty. If $\gamma$ is too large, the intersection contains many (and maybe all) routes from $a$ to $b$, and not all of them will be a good choice. Assuming that we knew the optimal parameter $\gamma_{\mathrm{OPT}}$, we could pick a route from $A_{\gamma_{\mathrm{OPT}}}(T_1) \cap A_{\gamma_{\mathrm{OPT}}}(T_2)$ at random. Buhmann et al. [4] suggest to set $\gamma_{\mathrm{OPT}}$ to the value $\gamma$ that maximizes the so-called *similarity*

$$S_\gamma = \frac{|A_\gamma(T_1) \cap A_\gamma(T_2)|}{|A_\gamma(T_1)||A_\gamma(T_2)|}. \tag{2}$$

Notice that up to now we did not consider how often a route is realized by a journey in a recorded timetable. This is undesirable from a practical point of view: when we pick a route from $A_{\gamma_{\mathrm{OPT}}}(T_1) \cap A_{\gamma_{\mathrm{OPT}}}(T_2)$ at random, the probability to obtain a route should depend on how frequently it is realized. Therefore we change the definition of $A_\gamma(T)$ to a *multiset* of routes, and $A_\gamma(T)$ contains a route $r$ as often as it is realized by a journey starting at time $t_A - \gamma$ or later, and arriving at time $t_A$ or earlier. Figure 3 shows an example with five lines $\{1,\ldots,5\}$ and two routes $r_1 = \langle 1,2,3\rangle$ and $r_2 = \langle 4,5\rangle$. We have $\mu^T_\gamma(r_1) = 3$: taking the second 1 and the second 2 (from above) as well as taking the third 1 and the second 2 are counted as different journeys since the departure times at $a$ differ. On the other hand, by our definition of journey we have to take the first occurence of a line that arrives, thus taking the first 1 and waiting for the second 2 is *not* counted.

Now the approximation set $A_\gamma(T)$ can be represented by a function $\mu^T_\gamma : \mathcal{R} \to \mathbb{N}_0$, where for a route $r \in \mathcal{R}$, $\mu^T_\gamma(r)$ is the number of journeys starting at time $t_A - \gamma$ or later, arriving at time $t_A$ or earlier and following the route $r$. Thus, we have $|A_\gamma(T)| = \sum_{r\in\mathcal{R}} \mu^T_\gamma(r)$, and for two recorded timetables $T_1, T_2$, we need to compute

$$\gamma_{\mathrm{OPT}} = \arg\max_\gamma \frac{\sum_{r\in\mathcal{R}} \min(\mu^{T_1}_\gamma(r), \mu^{T_2}_\gamma(r))}{\left(\sum_{r\in\mathcal{R}} \mu^{T_1}_\gamma(r)\right) \cdot \left(\sum_{r\in\mathcal{R}} \mu^{T_2}_\gamma(r)\right)}. \tag{3}$$

After computing the value $\gamma_{\text{OPT}}$, we pick a route $r$ from $A_{\gamma_{\text{OPT}}}(T_1) \cap A_{\gamma_{\text{OPT}}}(T_2)$ at random according to the probability distribution defined by

$$p_r := \frac{\min(\mu^{T_1}_{\gamma_{\text{OPT}}}(r), \mu^{T_2}_{\gamma_{\text{OPT}}}(r))}{\sum_{r \in \mathcal{R}} \min(\mu^{T_1}_{\gamma_{\text{OPT}}}(r), \mu^{T_2}_{\gamma_{\text{OPT}}}(r))}, \tag{4}$$

and search in the planned timetable for a journey from $a$ to $b$ that departs at time $t_A - \gamma_{\text{OPT}}$ or earlier, and that arrives at time $t_A$ or earlier.

**Computing the similarity** For $i \in \{1, 2\}$, we represent the function $\mu^{T_i}_{\gamma}$ by an $|\mathcal{R}|$-dimensional vector $\mu_i$ such that $\mu_i[r] = \mu^{T_i}_{\gamma}(r)$ for every $r \in \mathcal{R}$. We can compute the value $\gamma_{\text{OPT}}$ by a simple extension of the aforementioned sweepline algorithm. The modified algorithm again starts at time $t_A$, and considers all trips in $T_1$ and $T_2$ in reverse chronological order. The sweepline stops at every time when one or more trips in $T_1$ or in $T_2$ depart. Assume that the sweepline stops at time $t_A - \gamma$, and assume that it stopped at time $t_A - \gamma' > t_A - \gamma$ in the previous step. Of course, we have $\mu^{T_i}_{\gamma}(r) \geq \mu^{T_i}_{\gamma'}(r)$ for every $r \in \mathcal{R}$ and $i \in \{1, 2\}$. Let $\tau_1, \dots, \tau_k$ be the trips that depart in $T_1$ or $T_2$ at time $t_A - \gamma$. The idea is to compute the values of $\mu_i$ (representing $\mu^{T_i}_{\gamma}$) from the values computed in the previous step (representing $\mu^{T_i}_{\gamma'}$). This can be done as follows: for every trip $\tau_j$ occuring in $T_i$ and departing at time $t_A - \gamma$, we check whether there exists a route $r \in \mathcal{R}$ starting with $L(\tau_j)$. If yes, we distinguish two cases:

1. If $\mu_i[r] = 0$, then $\mu^{T_i}_{\gamma'}(r) = 0$, thus $r \notin A_{\gamma'}(T_i)$. If there exists a journey from $a$ to $b$ along $r$ departing at time $t_A - \gamma$ or later, and arriving at time $t_A$ or earlier, then $A_{\gamma}(T_i)$ contains $r$ exactly once. Thus, if $\textsc{Earliest-Arrival}(a, b, t_A - \gamma, r) \leq t_A$, we set $\mu_i[r] \leftarrow 1$.
2. If $\mu_i[r] > 0$, then $\mu^{T_i}_{\gamma'}(r) > 0$, thus $A_{\gamma'}(T_i)$ contains $r$ at least once. Thus, there exists a journey from $a$ to $b$ along $r$ departing at time $t_A - \gamma'$ or later, and arriving at time $t_A$ or earlier. Since $\tau_i$ is the only possibility to depart at $a$ between time $t_A - \gamma$ and $t_A - \gamma'$, $\tau_i$ is the first trip on a journey we never found before. Therefore it is sufficient to simply increase $\mu_i[r]$ by 1.

Up to now, we did not define when the algorithm terminates. In fact we stop if $\gamma$ exceeds a value $\gamma_{\text{MAX}}$. Let $t_A - \gamma_i$ be the starting time of an optimal journey in $T_i$. Of course, $\gamma_{\text{MAX}}$ has to be larger than $\max\{\gamma_1, \gamma_2\}$. In our experimental evaluation, we set $\gamma_{\text{MAX}}$ to be one hour before $t_A$; good choices for $\gamma_{\text{MAX}}$ will be investigated in further experiments.

## 5 Journey Reliability

**Success rate as reliability** Having several recorded timetables at our disposal, and a journey from $a$ to $b$, a natural approach to assess its *reliability* with respect to the given latest arrival time $t_A$ is to check how many times in the past the journey finished before $t_A$. Normalized by the total number of recorded timetables, we call this success rate the *coupled reliability*. This is the least information about robustness one would wish to obtain from online routing services when being presented, upon a query to the system, with a set of routes from $a$ to $b$.

**Few recorded timetables** The generalizing expressiveness of coupled reliability is limited (and biased towards outliers in the samples) if the number of recorded timetables is small. If lines in the considered transportation network suffer from delays (mostly) independently, we can heuristically extract from each of the $m$ given recorded timetables $T_1, \dots, T_m$ an individual timetable $T(i, l)$ for every line $l$ (storing just the travelled times of the specific line $l$ in timetable $T_i$), and then evaluate the considered journey on every relevant combination of these individual *decoupled* timetables. This enlarges the number of evaluations of the journey and thus has a chance to better generalize/express the observed travel times as typical situation.

**Decoupling the timetables** We can formally describe this process as follows. We consider $m$ recorded timetables $T_1, \ldots, T_m$, and we consider a journey $J$ from stop $a$ to stop $b$, specified by a departure time $t_D$, by a sequence of lines $\langle l_1, \ldots, l_k \rangle$, and by a sequence of transfer stops $\langle s_{\text{CH}}^{(1)}, \ldots, s_{\text{CH}}^{(k-1)} \rangle$.

We say that journey $J$ is *realizable in* $\langle T(i_1, l_1), T(i_2, l_2), \ldots, T(i_k, l_k) \rangle$, $i_1, \ldots, i_k \in \{1, \ldots, m\}$, *with respect to a given latest arrival time* $t_A$, if for every line $l_j$ there exists a trip $t_j$ (of the line $l_j$) in $T(i_j, l_j)$ such that

1. The departure time of trip $t_1$ from stop $a$ is after $t_D$,
2. the arrival time of trip $t_k$ at stop $b$ is before $t_A$, and
3. for every $j = 1, \ldots, k-1$, the arrival time of trip $t_j$ at stop $s_{\text{CH}}^{(j)}$ is before the departure time of trip $t_{j+1}$ at the same stop.

**Decoupled reliability** Clearly, there are $m^k$ ways to create a $k$-tuple $\langle T(i_1, l_1), \ldots, T(i_k, l_k) \rangle$. Let $M$ denote the number of those $k$-tuples in which journey $J$ is realizable with respect to a given $t_A$. We call the ratio $\frac{M}{m^k}$ the *decoupled reliability* of journey $J$ with respect to the latest arrival time $t_A$.

**Computational issues** Computing the coupled reliability is very easy: For every timetable $T_i \in \{T_1, \ldots, T_m\}$ we need to check whether the journey in question finished before time $t_A$ or not. This can be done by a simple linear time algorithm that simply "simulates" the journey in the timetable $T_i$, and checks whether the arrival time of the journey lies before or after $t_A$. The computation of decoupled reliability is not so trivial anymore, as the straightforward approach would require to enumerate all $m^k$ $k$-tuples $\langle T(i_1, l_1), \ldots, T(i_k, l_k) \rangle$, and thus an exponential time. In the following section, we present an algorithm that avoids such an exponential enumeration.

**Computing decoupled reliability** We can reduce the enumeration of all $k$-tuples $\langle T(i_1, l_1), T(i_2, l_2), \ldots, T(i_k, l_k) \rangle$ by observing that the linear order of the lines in journey $J$ allows to use dynamic-programming. Let us denote for simplicity the boarding, transfer, and arrival stops of journey $J$ as $s_0, s_1, \ldots, s_k$, where $s_0 = a$, $s_k = b$, and $s_j = s_{\text{CH}}^{(j)}$ for $j = 1, \ldots, k-1$. For every stop $s_{j-1}$, $j = 1, \ldots, k$, we store for every time event $t$ of a departing trip $\tau$ of line $l_j$ (in any of the timetables $T_1, \ldots, T_m$) a "success rate" of the journey $J$: the fraction $SR[s_{j-1}, t]$ of all tuples $\langle T(i_j, l_j), \ldots, T(i_k, l_k) \rangle$ in which the sub-journey of $J$ from $s_{j-1}$ to $s_k$ starting at time $t$ is realizable. For time $t$ not being a departure event, we extend the definition and set $SR[s_{j-1}, t] := SR[s_{j-1}, t']$, where $t'$ is the nearest time in the future for which a departing event exists. Having this information for every $j$, the decoupled reliability of $J$ is then simply $SR[s_0, t_D]$.

We can compute $SR[s_{j-1}, t]$ in the order of decreasing values of $j$. We initially set $SR[s_k, t_A] = 1$ (denoting that the fraction of successful sub-journeys arriving in $s_k$ is 1, if the sub-journey starts in $s_k$ and before $t_A$). The dynamic-programming like fashion for computing $SR[s_{j-1}, t]$ at any time $t$ then follows from the following recurrence:

$$SR[s_{j-1}, t] = \frac{1}{m} \sum_{i=1}^{m} SR[s_j, t_i], \tag{5}$$

where $t_i$ is the earliest arrival time of line $l_j$ at stop $s_j$ if the line uses timetable $T_i$ and does not depart before time $t$ from $s_{j-1}$.

When implementing the algorithm, we can save the (otherwise linear) time computation of the values of $t_i$ from the recurrence by simply storing this value and updating if needed.

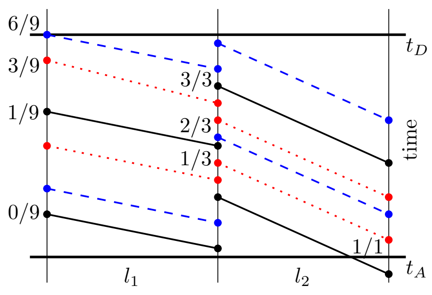


**Figure 4** A journey with two lines $l_1$ and $l_2$ and three timetables (solid black, dotted red, dashed blue). The fractions denote the stored values of $SR[s_j, t]$.

Figure 4 illustrates the algorithm, and the resulting decoupled reliability of 6/9. The running time of a naive implementation is $\mathcal{O}(k \cdot (m + e \log e))$, where $e$ is the maximum number of considered tram departing events at any station $s_j$.

## 6 Small Experimental Evaluation

In this section we describe and comment on a small experimental evaluation of the proposed approach to robust routing in public transportation networks. We first describe few observations/properties of our approach that serve as a kind of "mental" experiment. We have also implemented the proposed algorithms, and we report on our preliminary experiments with *real* public networks and *artificially* generated delays.

**Properties of the approach** Let $T_1$ and $T_2$ be two recorded timetables (from which we want to learn how to travel from stop $a$ to stop $b$ and arrive there before $t_A$). Consider the situation where the best journey $J$ to travel from $a$ to $b$ in timetable $T_1$ is the same as the best journey to travel from $a$ to $b$ in timetable $T_2$. Assuming that $T_1$ and $T_2$ represent typical delays, common sense dictates to use the very same journey $J$ also in the future. This is exactly what our approach does as well. Recall that $S_\gamma \leq 1$. Let $r$ be the route that corresponds to the journey $J$. In our case, setting $\gamma$ so that $A_\gamma(T_1) = A_\gamma(T_2) = \{r\}$, we get that $S_\gamma = \frac{|A_\gamma(T_1) \cap A_\gamma(T_2)|}{|A_\gamma(T_1)||A_\gamma(T_2)|} = 1$, and thus our approach computes the very same $\gamma$ and returns the journey $J$ as the recommendation to the user. These considerations can be generalized to the cases such as the one where $A_\gamma(T_1) = \{r\}$, $r \in A_\gamma(T_2)$, in which again $J$ will be returned as the recommendation to the user.

If only a reliable journey is required, and the travel time is not an issue, then suggesting to depart few days before $t_A$ is certainly sufficient. We now demonstrate that our approach does not work along these lines, and that it in fact reasonably balances the two goals *robustness* and *travel time*. We consider the symmetric situation where both $|A_\gamma(T_1)|$ and $|A_\gamma(T_2)|$ grow with $\gamma$ in the same way, i.e., for every $\gamma$, $|A_\gamma(T_1)| = |A_\gamma(T_2)|$. Let us only consider discrete values of $\gamma$, and let $\gamma_1$ be the largest $\gamma$ for which $A_{\gamma_1}(T_1) \cap A_{\gamma_1}(T_2) = \emptyset$. Let $x = |A_{\gamma_1}(T_1)|$. Then, for every $\gamma > \gamma_1$, $S_\gamma = \frac{\Delta_\gamma}{(x+\Delta_\gamma)^2}$ for some values of $\Delta_\gamma$. Simple calculation shows that $S_\gamma$ is maximized for $\Delta_\gamma = x$. We can interpret $x$ as the number of failed routes (that would otherwise make it if no delays appear). Then, $S_\gamma$ is maximized at the point that allows for another $\Delta_\gamma = x$ routes to joint the approximation sets $A_\gamma(T_i)$. Thus, the more disturbed the timetables are, the more "backward" in time we need to search for a robust route.

**Experimental evaluation** We implemented the algorithms presented in the sections 2, 3 and 4 in Java 7. The experiments were performed on one core of an Intel Core i5-3470 CPU

**Table 1** Comparison of the described methods over 100 test cases.

| | on time | less than 5 min late | less than 10 min late | avg arrival time | avg earlier depart. than Opt in $T_3$ |
|---|---|---|---|---|---|
| Unexpected Similarity, pick u.a.r. | 88% | 95% | 97% | 7:54 | 3.14 |
| Unexp. Sim., pick max. # occurences | 89% | 94% | 97% | 7:54 | 3.22 |
| Optimum in $T$ | 31% | 48% | 60% | 8:07 | -7.9 |
| 2nd Optimum in $T$ | 49% | 64% | 76% | 7:57 | 2.14 |
| Opt. in $T$ + end buffer time | 41% | 57% | 70% | 8:03 | -3.26 |
| Buffer time 3 min | 55% | 71% | 83% | 7:59 | 0.02 |
| Buffer time 5 min | 66% | 81% | 88% | 7:56 | 4.43 |

clocked at 3.2 GHz with 4 GB of RAM running Debian Linux 7.0. We used the combined tram and bus network of Zurich as input. It has 611 stops and 90 different line IDs. In our experiments, the actual number of lines itself is much higher (471), since multiple lines may operate under the same ID (e.g., lines in opposite directions, or lines coming from or returning to the depot). The planned timetable $T$ that we used is the official one for the Zurich network. However, trips departing before 6 a.m. or after 10 p.m. were ignored (since the timetable is only valid for 24 hours, trips starting before and ending after midnight are virtually interrupted at midnight, leading to a large number of lines).

We set the latest arrival time $t_A$ to 8 a.m., and carefully chose a small set of problematic stops $S'$ where delays usually occur. Then we generated 100 pairs of stops $(a, b)$ uniformly at random. For each pair, we generated three timetables $T_1$, $T_2$ and $T_3$ from $T$ by delaying every trip $\tau$ in $T$ between 0 and 3 minutes at every station $s \in S'$ (if $s$ occurs on $\tau$). These delays are 0 or 3 minutes with probability 1/8, and 1 or 2 minutes with probability 3/8. $T_1$ and $T_2$ are used as input to the algorithm, and the arrival time of the computed journey is measured in $T_3$. We use the following methods for computing the journey.

1. **Maximizing the Unexpected Similarity** Compute a route using the approach described in section 4. We consider two ways to pick a route from the intersection: 1) choose uniformly at random; 2) Choose the one with the maximum number of occurrences.
2. **Optimum in $T$** Find the best or the second best journey according to the planned timetable $T$. Compute also the latest journey arriving in $T$ five minutes before $t_A$.
3. **Buffer time for transfers** Consider the latest journey from $a$ to $b$ that arrives on time in $T$ such that at each transfer stop it have to wait for an additional "buffer time". We experiment with buffer times of 1 – 5 minutes.

For each of these statistics, we computed the following numbers (see Table 1): Percentage of the experiments where the proposed journey arrives on time, how often it arrives at most 5 minutes late, and how often it arrives at most 10 minutes late. We also computed the average arrival time of the journeys proposed by each method as well as the average difference between the departure time of the proposed journey to the optimal journey in $T_3$.

The average time for computing the optimum solution is 127ms, the time to compute a robust journey by using Unexpected Similarity is 262ms. We observed that our algorithm produces journeys that are on time in high percentage of cases, and on average we propose to depart only around 3 minutes earlier than the optimum in $T_3$, thus the cost we pay for this robustness is quite low. In comparison, the other considered approaches achieve much lower success rates. Even the generous buffer time of 5 minutes turns out not to be enough to beat our approach, which is rather surprising given the small delays in the considered timetables.

## 7 Discussion

We presented a novel framework for robust routing in frequent and dense urban public transportation networks based on observations of past traffic data. We introduced a new concept to describe a travel plan, a *journey*, that is not only well suited for our robustness issues, but also represents a natural and convenient description for the traveler. We also provided a bag of algorithmic tools to handle this concept, tailored towards the proposed robustness measures. We described a simple way to assess the reliability of a given journey. We also used a different approach to robustness and described how to find a robust journey according to it. We are preparing further experiments to confirm efficiency of the presented algorithms and to evaluate the quality of the computed robust journeys.

Future work is to examine how the described methods can be extended to support a fully multi-modal scenario, e.g., how to integrate walking. We believe that the modelling itself is easy, while the performance of the algorithms will decrease significantly unless we develop special techniques. Also considering and exploring different robustness concepts for journeys may be worthwhile.

### References

1 Hannah Bast, Erik Carlsson, Arno Eigenwillig, Robert Geisberger, Chris Harrelson, Veselin Raychev, and Fabien Viger. Fast routing in very large public transportation networks using transfer patterns. In Mark Berg and Ulrich Meyer, editors, *Algorithms – ESA 2010*, volume 6346 of *LNCS*, pages 290–301. Springer Berlin Heidelberg, 2010.

2 Reinhard Bauer, Daniel Delling, and Dorothea Wagner. Experimental study of speed up techniques for timetable information systems. *Networks*, 57(1):38–52, 2011.

3 Justin Boyan and Michael Mitzenmacher. Improved results for route planning in stochastic transportation. In *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 895–902. Society for Industrial and Applied Mathematics, 2001.

4 Joachim M. Buhmann, Matúš Mihalák, Rastislav Šrámek, and Peter Widmayer. Robust optimization in the presence of uncertainty. In Robert D. Kleinberg, editor, *ITCS*, pages 505–514. ACM, 2013.

5 Daniel Delling, Thomas Pajor, and Dorothea Wagner. Engineering time-expanded graphs for faster timetable information. In Ravindra K. Ahuja, Rolf H. Möhring, and Christos D. Zaroliagis, editors, *Robust and Online Large-Scale Optimization*, volume 5868 of *LNCS*, pages 182–206. Springer Berlin Heidelberg, 2009.

6 Daniel Delling, Thomas Pajor, and Renato F Werneck. Round-based public transit routing. *Algorithm Engineering and Experiments (ALENEX)*, pages 130–140, 2012.

7 Julian Dibbelt, Thomas Pajor, Ben Strasser, and Dorothea Wagner. Intriguingly simple and fast transit routing. In Vincenzo Bonifaci, Camil Demetrescu, and Alberto Marchetti-Spaccamela, editors, *SEA*, volume 7933 of *LNCS*, pages 43–54. Springer, 2013.

8 Yann Disser, Matthias Müller-Hannemann, and Mathias Schnee. Multi-criteria shortest paths in time-dependent train networks. In *Experimental Algorithms*, pages 347–361. Springer, 2008.

9 H Frank. Shortest paths in probabilistic graphs. *Operations Research*, 17(4):583–599, 1969.

10 Marc Goerigk, Martin Knoth, Matthias Müller-Hannemann, Marie Schmidt, and Anita Schöbel. The price of robustness in timetable information. In *11th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems*, pages 76–87, 2011.

11 Matthias Müller-Hannemann and Mathias Schnee. Efficient timetable information in the presence of delays. In Ravindra K. Ahuja, Rolf H. Möhring, and Christos D. Zaroliagis,

editors, *Robust and Online Large-Scale Optimization*, volume 5868 of *LNCS*, pages 249–272. Springer, 2009.

12 Matthias Müller-Hannemann, Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. Timetable information: Models and algorithms. In *Algorithmic Methods for Railway Optimization*, pages 67–90. Springer, 2007.

13 Evdokia Nikolova, Jonathan A Kelner, Matthew Brand, and Michael Mitzenmacher. Stochastic shortest paths via quasi-convex maximization. In *Algorithms–ESA 2006*, pages 552–563. Springer, 2006.

14 Evangelia Pyrga, Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. Efficient models for timetable information in public transportation systems. *Journal of Experimental Algorithmics (JEA)*, 12:2–4, 2008.

# Robust Routing in Urban Public Transportation: How to Find Reliable Journeys Based on Past Observations *

**Kateřina Böhmová, Matúš Mihalák, Tobias Pröger, Rastislav Šrámek, and Peter Widmayer**

**Institute of Theoretical Computer Science, ETH Zurich, Switzerland**
`{kboehmov,mmihalak,tproeger,rsramek,widmayer}@inf.ethz.ch`

## Abstract

We study the problem of robust routing in urban public transportation networks. In order to propose solutions that are robust for typical delays, we assume that we have past observations of real traffic situations available. In particular, we assume that we have "daily records" containing the observed travel times in the whole network for a few past days. We introduce a new concept to express a solution that is feasible in any record of a given public transportation network. We adapt the method of Buhmann et al. [4] for optimization under uncertainty, and develop algorithms that allow its application for finding a robust journey from a given source to a given destination. The performance of the algorithms and the quality of the predicted journey are evaluated in a preliminary experimental study. We furthermore introduce a measure of reliability of a given journey, and develop algorithms for its computation. The robust routing concepts presented in this work are suited specially for public transportation networks of large cities that lack clear hierarchical structure and contain services that run with high frequencies.

**1998 ACM Subject Classification** F.2.2 Nonnumerical Algorithms and Problems, G.2.2 Graph Theory (Graph algorithms, Network problems), I.2.6 Learning

**Keywords and phrases** Algorithms, Optimization, Robustness, Route planning, Public transportation

**Digital Object Identifier** 10.4230/OASIcs.ATMOS.2013.27

## 1 Introduction

We study the problem of routing in urban public transportation networks, such as tram and bus networks in large cities, focusing on the omnipresent uncertain situations when (typical) delays occur. In particular, we search for robust routes that allow reliable yet quick passenger transportation. We think of a "dense" tram network in a large city containing many tram lines, where each tram line is a sequence of stops that is served repeatedly during the day, and where there are several options to get from one location to another. Such a network usually does not contain clear hierarchical structure (as opposed to train networks), and each line is served with high frequency. Given two tram stops $a$ and $b$ together with a latest arrival time $t_A$, our goal is to provide a simple yet robust description of how to travel in the

* This work has been partially supported by the Swiss National Science Foundation (SNF) under the grant number 200021 138117/1, and by the EU FP7/2007-2013 (DG CONNECT.H5-Smart Cities and Sustainability), under grant agreement no. 288094 (project eCOMPASS). Kateřina Böhmová is a recipient of the Google Europe Fellowship in Optimization Algorithms, and this research is supported in part by this Google Fellowship.

given network from $a$ to $b$ in order to arrive on time $t_A$ even in the presence of typical delays. We base our robustness concepts on past traffic data in a form of recorded timetables – the actually observed travel times of all lines in the course of several past days. If no delays occur, such a recorded timetable corresponds to the scheduled timetable for that day.

The standard approach to describe a travel plan from $a$ to $b$ in a given tram network is to specify, according to a scheduled timetable, the concrete sequence of vehicles together with transfer stops and departure/arrival times for each transfer stop. Such a travel plan may look like this: Take the tram 6 at 12:33 from stop $a$ and leave it at 12:47 at transfer stop $s$; then take the tram 10 at 12:51 from $s$ and leave it at 12:58 at $b$. However, such a travel plan may become infeasible on a concrete day due to delays: Imagine a situation where the tram 6 left $a$ at 12:33, but arrived to $s$ only at 12:53, and the tram 10 leaving $s$ at 12:51 was on time. Then, the described travel plan would bring the passenger to stop $s$ but it does not specify how to proceed further in order to arrive to $b$.

We observe that the standard solution concepts (such as paths in a time-expanded graph) are not suitable for our setting. We introduce a new concept to express a solution, which we call a *journey*, that is feasible in any recorded timetable of a given transportation network assuming the timetable to be periodic. A journey specifies an initial time $t_D$ and then only a sequence of transportation lines $\langle l_1(\text{tram}), l_2(\text{bus}), \ldots, l_k(\text{tram})\rangle$ together with transfer stops $\langle s_1, \ldots, s_{k-1}\rangle$. This travel plan suggests to start waiting at $a$ at time $t_D$, take the first tram of line $l_1$ that comes and travel to stop $s_1$, then change to the first coming bus of line $l_2$, etc. Since we assume that the frequency of vehicles serving each line is high, such a travel plan is not only feasible in our setting but also reasonable, and provides the passenger with all the necessary information. We provide algorithms to efficiently compute these journeys.

Equipped with the introduced solution concept of a journey, we can easily adapt the method of Buhmann et al. [4] for optimization under uncertainty, and apply it to identify robust travel plans. A key ingredient of the method is the ability to count the number of (possibly exponentially many) "good" solutions. Our solution concept allows us to develop efficient algorithms to compute the number of all journeys from $a$ to $b$ that depart after the time $t_D$ and arrive before the time $t_A$.

Finally, we suggest an alternative simple measure for reliability of a given journey, expressed simply as the fraction of recorded timetables where the journey was successful and allowed to arrive at the destination on time. We provide efficient algorithms for computation of this measure.

## 2 Related Work

The problem of finding a fastest journey (according to the planned timetable) using public transportation has been extensively studied in the literature. Common approaches model the transportation network as a graph and compute a shortest path in this graph (see [12] for a survey). Various improvements have been developed, and experimental studies suggest that these can also be used in practice (see, e.g., [2, 5, 14]). Recent approaches avoid the construction of a graph and process the timetable directly [6, 7]. For example, Delling et al. [6] describe an approach which is centered around transportation lines (such as train or bus lines) and which can be used to find all pareto-optimal journeys when the arrival time and the number of stops are considered as criteria. Bast et al. [1] observe that for two given stops, we can find and encode each sequence of intermediate transfer stations (i.e., stations where we change from one line to another) that can lead to an optimal route. The set of these sequences of transfers is called *transfer pattern*. These patterns can be precomputed,

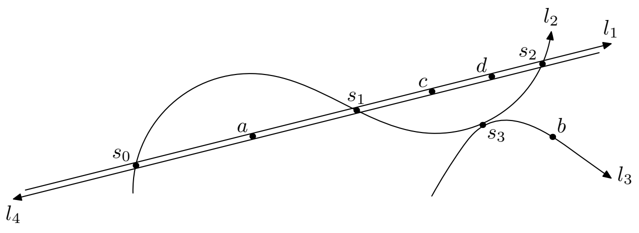


**Figure 1** The line $l_1$ is a sequence of stops $\langle \dots, s_0, a, s_1, c, d, s_2, \dots \rangle$. The line $l_4 = \langle \dots, s_2, d, c, s_1, b, s_0, \dots \rangle$ that goes in the opposite direction to $l_1$ is considered to be a different line. In this example, both $a \lhd l_1$ and $a \lhd l_4$ hold, but $a \lhd l_2$ does not. Similarly, $s_1 \lhd s_2 \lhd l_1$ holds, but $s_1 \lhd s_2 \lhd l_4$ does not. The set $l_1 \cap l_2$ of all stops common to $l_1$ and $l_2$ is $\{s_0, s_1, s_2\}$. Moreover, when travelling from $a$ to $b$ using a route $\langle l_1, l_2, l_3 \rangle$, this network is an example where not every stop in $l_1 \cap l_2$ is suitable for changing from $l_1$ to $l_2$: We cannot choose $s_0$ as transfer stop since it is served before $a$. If $s_2$ was chosen, then $l_3$ can never be reached without travelling back. Thus, the only valid stop to change the line is $s_1$.

leading to very fast query times. These approaches are similar to our approach in the sense that they try to explicitly exploit the problem structure (e.g., by considering lines) instead of implicitly modelling all properties into a graph.

For computing *robust* journeys in public transportation, stochastic networks have been studied [3, 9, 13], where the delays between successive edges are random variables. Dibbelt et al. [7] study the case when stochastic delays on the vehicles are given. In a situation when timetables are fixed, Disser et al. [8] used a generalization of Dijkstra's algorithm to compute pareto-optimal multi-criteria journeys. They define the reliability of a journey as a function depending on the minimal time to change between two subsequent trains, and use it as an additional criterion. Müller-Hannemann and Schnee [11] introduced the concept of a *dependency graph* for a prediction of secondary delays caused by some current primary delays, which are given as input. They also show how to compute a journey that is optimal with respect to the predicted delays. Goerigk et al. [10] consider a given set of delay *scenarios* for every event, and adapt *strict robustness* to it, i.e. they aim to compute a journey that arrives on time for every scenario. Furthermore, the concept of *light robustness* is introduced, which aims to compute a journey that maximizes the number of scenarios in which the travel time of this journey lies at most a fixed time above the optimum. Strict robustness requires a feasible solution for every realization of delays for every event. This is quite conservative, as in reality not every combination of event delays appears. Our approach tries to avoid this by learning from the typical delay scenarios as recorded for each individual day.

## 3 Modeling issues

### 3.1 Model

**Stops and lines** Let $\mathcal{S}$ be a set of stops, and $\mathcal{L} \subset \bigcup_{i=2}^{|\mathcal{S}|} \mathcal{S}^i$ be a set of lines (e.g., bus lines, tram lines or lines of other means of transportation). The following basic definitions are illustrated in Figure 1. Every line $l \in \mathcal{L}$ is a sequence of $S(l)$ stops $\langle s_1^{(l)}, \dots, s_{S(l)}^{(l)} \rangle$, where, for every $i \in \{1, \dots, S(l) - 1\}$, the stop $s_i^{(l)}$ is served directly before $s_{i+1}^{(l)}$ by the line $l$. We explicitly distinguish two lines that serve the same stops but have opposite directions (these may be operated under the same identifier in reality). For a stop $s \in \mathcal{S}$ and a line $l \in \mathcal{L}$, we write $s \lhd l$ if $s$ is a stop on the line $l$, i.e. if there exists an index $i \in \{1, \dots, S(l)\}$ such that $s = s_i^{(l)}$. Furthermore, for two stops $s_1, s_2 \in \mathcal{S}$ and a line $l \in \mathcal{L}$ we write $s_1 \lhd s_2 \lhd l$ if both $s_1$ and $s_2$ are stops on $l$ and $s_1$ is served before $s_2$, i.e. if there exist indices $i, j \in \mathbb{N}$, $1 \leq i \leq S(l) - 1$, $i + 1 \leq j \leq S(l)$ such that $s_1 = s_i^{(l)}$ and $s_2 = s_j^{(l)}$. For two lines $l_1, l_2 \in \mathcal{L}$, we define $l_1 \cap l_2$ to be the set of all stops $s \in \mathcal{S}$ that are served both by $l_1$ and $l_2$.

**Trips and timetables** While the only information associated with a line itself are its consecutive stops, it usually is operated multiple times per day. Each of these concrete realizations that departs at a given time of the day is called a *trip*. With every trip $\tau$ we associate a line $L(\tau) \in \mathcal{L}$. By $L^{-1}(l)$ we denote the set of *all* trips associated with a line $l \in \mathcal{L}$. For a trip $\tau$ and a stop $s \in \mathcal{S}$, let $A(\tau, s)$ be the arrival time of $\tau$ at stop $s$, if $s \lhd L(\tau)$. Analogously, let $D(\tau, s)$ be the departure time of $\tau$ at $s$. In the following, we assume time to be modelled by integers. For a given trip $\tau$, we require $A(\tau, s) \leq D(\tau, s)$ for every stop $s \in L(\tau)$. Furthermore we require $D(\tau, s_1) \leq A(\tau, s_2)$ for every two stops $s_1, s_2 \in \mathcal{S}$ with $s_1 \lhd s_2 \lhd L(\tau)$. A set of trips is called a *timetable*. We distinguish between

1. the *planned* timetable $T$. We assume it to be periodic, i.e., every line realized by some trip $\tau$ will be realized by a later trip $\tau'$ again (probably not on the same day).
2. *recorded* timetables $T_i$ that describe how various lines were operated during a given time period (i.e., on a concrete day or during a concrete week). These recorded timetables are concrete executions of the planned timetable.

In the following, *timetable* refers both to the planned as well as to a recorded timetable.

**Goal** In the following, let $a, b \in \mathcal{S}$ be two stops, $m \in \mathbb{N}_0$ be the maximal allowed number of line changes, and $t_A \in \mathbb{N}$ be the latest arrival time. A *journey* consists of a departure time $t_D$, a sequence of lines $\langle l_1, \ldots, l_k \rangle$, $k \leq m + 1$ and a sequence of transfer stops $\langle s_{\text{CH}}^{(1)}, \ldots, s_{\text{CH}}^{(k-1)} \rangle$. The intuitive interpretation of such a journey is to start at stop $a$ at time $t_D$, take the first line $l_1$ (more precisely, the first available trip of the line $l_1$), and for every $i \in \{1, \ldots, k-1\}$, leave $l_i$ at stop $s_{\text{CH}}^{(i)}$ and take the next arriving line $l_{i+1}$ immediately. Our goal is to compute a recommendation to the user in form of one or more (robust) journeys from $a$ to $b$ that will likely arrive on time (i.e., before time $t_A$) on a day for which the concrete travel times are not known yet. We formalize the notion of robustness later. We note that for the convenience of the user, one should handle two different lines $l_1$ and $l_2$ operating between two stops $s_1$ and $s_2$ as one (virtual) line, and provide recommendations of the form "in $s_1$, take the first line $l_1$ or $l_2$ to $s_2$, etc.". For the sake of simplicity we do not pursue this generalization further, but will consider this in the future.

**Routes** Let $k \in \{1, \ldots, m + 1\}$ be an integer. A sequence of lines $r = \langle l_1, \ldots, l_k \rangle \in \mathcal{L}^k$ is called a *feasible route from $a$ to $b$* if there exist $k + 1$ stops $s_0 := a, s_1, \ldots, s_{k-1}, s_k := b$ such that $s_{i-1} \lhd s_i \lhd l_i$ for every $i \in \{1, \ldots, k\}$, i.e., if both $s_{i-1}$ and $s_i$ are stops on line $l_i$, and $s_{i-1}$ is served before $s_i$ on line $l_i$. Notice that on a feasible route $r \in \mathcal{L}^k$ we need to change the line at $k - 1$ transfer stops. Let

$$\mathcal{R}_{ab}^m = \{r \in \mathcal{L} \cup \mathcal{L}^2 \cup \cdots \cup \mathcal{L}^{m+1} \mid r \text{ is a feasible route from } a \text{ to } b\} \tag{1}$$

be the set of all feasible routes from $a$ to $b$ using at most $m$ transfer stops. If $a$, $b$ and $m$ are clear from the context, for simplicity we just write $\mathcal{R}$ instead of $\mathcal{R}_{ab}^m$. Notice that by definition, a line $l$ may occur multiple times in a route. This is reasonable because there might be two transfer stops $s, s'$ on $l$ and one or more intermediate lines that travel faster from $s$ to $s'$ than $l$ does. Additionally, notice that a route does not contain *any* time information.

## 3.2 Computation of Feasible Routes

**Input data** In this section we describe an algorithm that, given a set of stops $\mathcal{S}$ and a set of lines $\mathcal{L}$, finds the set $\mathcal{R}$ of all feasible routes that allow to travel from a given initial stop $a$ to a given destination stop $b$ using at most $m$ transfer stops (also called transfers). Notice that to compute $\mathcal{R}$ we only need the network structure, no particular timetable is necessary.

**Preprocessing the input** We preprocess the input data and construct data structures to allow efficient queries of the following types:

1. $Q(l, s)$: *Compute the position of $s$ on $l$.* Given a line $l = \langle s_1, \ldots, s_{|S(l)|}\rangle$, $Q(l, s)$ returns $j$ if $s$ is the $j$-th stop on $l$, i.e., if $s = s_j$, or 0 if $s$ is not served by $l$.
2. $Q(l, s_i, s_j)$: *Determine whether $s_i$ is served before $s_j$ on $l$.* Given a line $l$ and two stops $s_i, s_j$, $Q(l, s_i, s_j)$ returns `TRUE` iff $s_i, s_j \triangleleft l$ and $s_i \triangleleft s_j \triangleleft l$.
3. $Q(l_i, l_j)$: *Determine $l_i \cap l_j$ (i.e., the stops shared by $l_i$ and $l_j$) in a compact, ordered format.* Given two lines $l_i$, and $l_j$, $Q(l_i, l_j)$ returns the set $l_i \cap l_j$ of stops shared by these lines. We encode $l_i \cap l_j$ into an ordered set $I_{ij}$ of pairs of stops with respect to $l_i$ in such a way that $(s_q, s_r) \in I_{ij}$ indicates that $l_i$ and $l_j$ share the stops $s_q$, $s_r$, and all the stops in between on the line $l_i$ (independent of their order on $l_j$). Thus, $Q(l_i, l_j)$ outputs the described sorted set $I_{ij}$ of pairs of stops. The motivation to compress $l_i \cap l_j$ into $I_{ij}$ is that, in practice, there may be many stops shared by $l_i$ and $l_j$, but only a small number of contiguous intervals of such stops. Notice that $Q(l_i, l_j)$ doesn't need to be equal to $Q(l_j, l_i)$, nor the sequence in reverse order; an example is given in Figure 2.

Notice that these queries can be answered in expected constant time if implemented using suitable arrays or hashing tables.

**Graph of line incidences** The function $Q(l_i, l_j)$ induces the following directed graph $G$. The set $V$ of vertices of $G$ corresponds to the set of lines $\mathcal{L}$. There is an edge from a vertex (line) $l_i$ to a vertex $l_j$ if and only if $Q(l_i, l_j) \neq \emptyset$. Then, $Q(l_i, l_j)$ is represented as a tag of the edge $(l_i, l_j)$. We construct and represent the graph $G$ as adjacency lists.

**Preliminary observations** Given two stops $a$ and $b$, and a number $m$, we want to find all routes $\mathcal{R}$ that allow to travel from $a$ to $b$ using at most $m$ transfers in the given public transportation network described by a set of stops $\mathcal{S}$ and a set of lines $\mathcal{L}$. Notice that each such route $r = \langle l_1, \ldots, l_k\rangle \in \mathcal{L}^k$ with $0 < k \leq m + 1$ has the following properties.

1. Both $Q(l_1, a)$ and $Q(l_k, b)$ are nonzero (i.e., $a \triangleleft l_1$, and $b \triangleleft l_k$).
2. The vertices $l_1, \ldots, l_k$ form a path in $G$ (i.e., $l_i \cap l_{i+1} \neq \emptyset$ for every $i = 1, \ldots, k - 1$).
3. There exists a sequence of stops $a = s_0, s_1, \ldots, s_{k-1}, s_k = b$ such that $Q(l_i, s_{i-1}, s_i)$ is `TRUE` (i.e., $s_{i-1} \triangleleft s_i \triangleleft l_i$) for every $i = 1, \ldots, k$.

These observations lead to the following algorithm to find the set of routes $\mathcal{R}$.

**All routes algorithm** For the stop $a$, determine the set $\mathcal{L}_a$ of all lines passing through $a$. Then explore the graph $G$ from the set $\mathcal{L}_a$ of vertices in the following fashion. For each vertex $l_1 \in \mathcal{L}_a$, perform a depth-first search in $G$ up to the depth $m$, but do not stop when finding a vertex that has already been found earlier. In each step, try to extend a partial path $\langle l_1, \ldots, l_j\rangle$ to a neighbor $l'_j$ of $l_j$ in $G$. Keep track of the *current transfer stop* $s_q$. This is a stop on the currently considered line $l_j$ such that $s_q$ is the stop with the smallest position on $l_j$ at which it is possible to transfer from $l_{j-1}$ to $l_j$, considering the partial path from $l_1$ to $l_{j-1}$. In other words, $s_q$ is the stop on the considered route where the line $l_j$ can be boarded. Each step of the algorithm is characterized by a *search state*: a partial path $P = \langle l_1, \ldots, l_j\rangle$, and a current transfer stop $s_q$ that allowed the transfer to line $l_j$. The initial search state consists of the partial path $P = \langle l_1\rangle$ and the current transfer stop $a$. More specifically, to

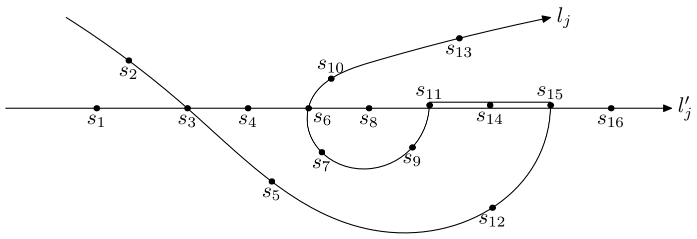


**Figure 2** Lines $l_j$ and $l'_j$ have common stops $s_3$, $s_6$, $s_{11}$, $s_{14}$, and $s_{15}$. The ordered set $I_{jj'} = Q(l_j, l'_j)$ consists of the pairs $\{(s_3, s_3), (s_{15}, s_{11}), (s_6, s_6)\}$. Thus, the last stop in the last interval of $I_{jj'}$ is the stop $s_6$. On the other hand, the ordered set $I_{j'j} = Q(l'_j, l_j)$ consists of the pairs $\{(s_3, s_3), (s_6, s_6), (s_{11}, s_{15})\}$. Now, imagine that the current transfer stop $s_q$ for a partial path $P = \langle l_1, \ldots, l_j \rangle$ is $s_2$, then the stop $s_3$ is the current transfer stop $s'_q$ for a partial path $P' = \langle l_1, \ldots, l_j, l'_j \rangle$. However, observe that if $s_q$ is $s_{12}$, then $s'_q$ needs to be $s_6$.

process a search state with the partial path $P = \langle l_1, \ldots, l_j \rangle$, and the current transfer stop $s_q$, perform the following tasks:

1. Check whether the line corresponding to the vertex $l_j$ contains the stop $b$ and whether $s_q$ is before $b$ on $l_j$. If this is the case (i.e., the query $Q(l_j, s_q, b)$ returns `TRUE`), then the partial path $P$ corresponds to a feasible route and is output as one of the solutions in $\mathcal{R}$.
2. If the partial path $P$ contains at most $m-1$ edges (thus the corresponding route has at most $m-1$ transfers, and can be extended), then for each neighbor $l'_j$ of $l_j$ check whether extending $P$ by $l'_j$ is possible (and if so, update the current transfer stop) as follows. Let $I_{jj'} = Q(l_j, l'_j)$ be the set of pairs of stops sorted as described in the previous section. Recall that each pair $(s_u, s_v) \in I_{jj'}$ encodes an interval of one or several consecutive stops on $l_j$ that are also stops on the line $l'_j$. Let $s_z$ be the last stop in the last interval of $I_{jj'}$. Similarly, let $I_{j'j} = Q(l'_j, l_j)$. If $Q(l_j, s_q, s_z)$ is `TRUE`, then $s_q \lhd s_z \lhd l_j$, and the path $P$ can be extended to $l'_j$.
   a. We determine the current transfer stop $s'_q$ for $l'_j$ by considering the pairs/intervals of $I_{j'j}$ in ascending order and deciding whether the position of $s_q$ on the line $l_j$ is before one of the endpoints of the currently considered interval. We refer to Figure 2 for a nontrivial case of computing of the current transfer stop.
   b. Perform the depth search with the search state consisting of the partial path $P' = \langle l_1, \ldots, l_j, l'_j \rangle$ and the current transfer stop $s'_q$.

   Otherwise, if $Q(l_j, s_q, s_z)$ is `FALSE`, it is not possible to extend $P$ to $l'_j$.

The theoretical running time of the algorithm is $\mathcal{O}(\Delta^m)$, where $\Delta$ is the maximum degree of $G$. However, we believe that in practice the actual running time will rather linearly correspond to the size of the output $\mathcal{O}(m|\mathcal{R}|)$. On real-world data, the algorithm performs reasonably fast (see section 6 for details).

## 3.3 Computing the earliest arriving journey

**Recursive computation** As previously stated, let $a \in \mathcal{S}$ be the initial stop, $b \in \mathcal{S}$ be the destination stop, $\varepsilon(s, l, l')$ be the minimum time to change from line $l$ to line $l'$ at station $s$, and $t_A \in \mathbb{N}$ be the latest arrival time. In the previous section we showed how the set $\mathcal{R}$ of all feasible routes from $a$ to $b$ can be computed. However, instead of presenting just a route $r \in \mathcal{R}$ to the user, our final goal is to compute a departure time $t_0$ and a *journey* that arrives at $b$ before time $t_A$. For the following considerations, we assume the underlying timetable (either the planned or a recorded timetable) to be fixed. Given $a, b \in \mathcal{S}$, an initial departure time $t_0 \in \mathbb{N}$, and a route $r = \langle l_1, \ldots, l_k \rangle \in \mathcal{R}$, a journey along $r$ that arrives as early as possible can be computed as follows. We start at $a$ at time $t_0$ and take the first line $l_1$ that arrives. Then we compute an appropriate transfer stop $s \in l_1 \cap l_2$ (that is served both by $l_1$ as well as by $l_2$) and the arrival time $t_1$ at $s$, leave $l_1$ there and compute recursively the earliest arrival time when departing from $s$ at time at least $t_1 + \varepsilon(s, l_1, l_2)$, following the route $\langle l_2, \ldots, l_k \rangle$. Notice that the selection of an appropriate transfer stop $s$ is the only non-trivial part due to mainly two reasons:

1. The lines $l_1$ and $l_2$ may operate with different speeds (e.g., because $l_1$ is a fast tram while $l_2$ is a slow bus), or $l_1$ and $l_2$ separate at a stop $s_1$ and join later again at a stop $s_2$ but the overall travel times of $l_1$ and $l_2$ differ between $s_1$ and $s_2$. Depending on the situation, it may be better to leave $l_1$ as soon or as late as possible, or anywhere inbetween.
2. The lines $l_1$ and $l_2$ may separate at a stop $s_1$ and join later again at a stop $s_2$. If all transfer stops in $l_2 \cap l_3$ are served by $l_2$ before $s_2$, then leaving $l_1$ at $s_2$ is not an option since $l_3$ is not reachable anymore. See Figure 1 for a visualization.

The idea now is to find the earliest trip of line $l_1$ that departs from $a$ at time $t_0$ or later, iterate over all stops $s \in l_1 \cap l_2$, and compute recursively the earliest arrival time when continuing the journey from $s$ having a changing time of at least $\varepsilon(s, l_1, l_2)$. Finally, we return the smallest arrival time that was found in one of the recursive calls.

**Issues and improvement of the recursive algorithm** An issue with this naïve implementation is the running time, which might be exponential in $k$ in the worst-case (if $|l_i \cap l_{i+1}| > 1$ for $\Omega(k)$ many $i \in \{1, \ldots, k-1\}$). Let $\tau$ and $\tau'$ be two trips with $L(\tau) = L(\tau')$. If $\tau$ leaves before $\tau'$ at some stop $s$, we assume that it will never arrive later than $\tau'$ at any subsequent stop $s'$, $s \lhd s' \lhd L(\tau)$, i.e. consecutive trips of the same line do not overtake. For a line $l \in \mathcal{L}$ and a set of trips $T_l \subseteq L^{-1}(l)$, it follows that taking the earliest trip in $T_l$ never results in a later arrival at $b$ than taking any other trip from $T_l$. Furthermore, a trip $\tau \in T_l$ is operated earlier than a trip $\tau' \in T_l$ iff $A(\tau, s) < A(\tau', s)$ for *any* stop $s \lhd l$.

Thus, we can iterate over some appropriate stops in $l_1 \cap l_2$ to find the earliest reachable trip associated with $l_2$. We just need to ignore those stops where changing to $l_3$ is no longer possible (see Figure 1 for an example).

**Computing appropriate transfer stops** The problem to find these appropriate stops can be solved by first sorting $l_1 \cap l_2 = \{s_1, \ldots, s_n\}$ such that $s_j \lhd s_{j+1} \lhd l_1$ for every $j \in \{1, \ldots, n-1\}$. Obviously, all stops that appear before $a$ on line $l_1$ cannot be used for changing to $l_2$. This problem can easily be solved by considering only those stops $s_j$ where $a \lhd s_j \lhd l_1$. Unfortunately, the last $m \geq 0$ stops $s_{n-m+1}, \ldots, s_n$ might also not be suitable for changing to $l_2$ because they may prevent us later to change to some line $l_j$ (e.g., if *all* stops of $l_2 \cap l_3$ are served before $s_{n-g+1}, \ldots, s_n$ on $l_2$, then changing to $l_3$ is no longer possible). We solve this problem by precomputing (the index of) the last stop $s_j$ where all later lines are still reachable. This can be done backwards: we start at $b$, order the elements of $l_k \cap l_{k-1}$ as they appear on line $l_k$, and find the last stop that is served before $b$ on $l_k$. We recursively continue with $l_1, \ldots, l_{k-1}$ and use the stop previously computed as the stop that still needs to be reachable.

**Iterative algorithm** The improved algorithm first iterates over $i \in \{1, \ldots, k-1\}$, and uses the aforementioned algorithm to precompute the index last$[i]$ of the last stop where changing from $l_i$ to $l_{i+1}$ is still possible (with respect to the route $\langle l_1, \ldots, l_k \rangle$). After that, for every $i \in \{1, \ldots, k-1\}$, we iterate over the appropriate transfer stops $s \in l_i \cap l_{i+1}$ where changing to $l_{i+1}$ is possible, and find among those the stop $s_{\mathrm{CH}}^{(i)}$ where the earliest trip $\tau_{i+1}$ associated with line $l_{i+1}$ departs. Finally we obtain a sequence of trips $\tau_1, \ldots, \tau_k$ along with transfer stops $s_{\mathrm{CH}}^{(0)} := a, s_{\mathrm{CH}}^{(1)}, \ldots, s_{\mathrm{CH}}^{(k)}$ to change lines. Since we gradually compute the earliest trips $\tau_i$ for each of the lines $l_i$, the earliest time to arrive at $b$ is simply $A(\tau_k, b)$.

```
EarliestArrival($a, b, t_0, \langle l_1, \dots, l_k \rangle$)
1  $\text{last}[k] \leftarrow b$
2  for $i \leftarrow k, \dots, 2$ do
3      Order the elements of $l_i \cap l_{i-1} = \{s_1, \dots, s_n\}$ s.t. $s_j \lhd s_{j+1} \lhd l_{i-1}\ \forall j \in \{1, \dots, n-1\}$.
4      $\text{last}[i-1] \leftarrow \max\{j \in \{1, \dots, n\} \mid s_j \lhd \text{last}[i] \lhd l_i\}$
5  $\tau_1 \leftarrow \arg\min_{\tau \in L^{-1}(l_1)}\{D(\tau, a) \mid D(\tau, a) \geq t_0\}$;  $s_{\text{CH}}^{(0)} \leftarrow a$
6  for $i \leftarrow 1, \dots, k-1$ do
7      Order the elements of $l_i \cap l_{i+1} = \{s_1, \dots, s_n\}$ s.t. $s_j \lhd s_{j+1} \lhd l_i\ \forall j \in \{1, \dots, n-1\}$.
8      $\tau_{i+1} \leftarrow$ null;  $s_{\text{CH}}^{(i)} \leftarrow$ null;  $A_{s_n}^{(i+1)} \leftarrow \infty$
9      for $j \leftarrow 1, \dots, \text{last}[i]$ do
10         if $s_{\text{CH}}^{(i-1)} \lhd s_j \lhd l_i$ and $s_j \lhd \text{last}[i+1] \lhd l_{i+1}$ then
11             $\tau' \leftarrow \arg\min_{\tau \in L^{-1}(l_{i+1})}\{D(\tau, s_j) \mid D(\tau, s_j) \geq A(\tau_i, s_j) + \varepsilon(s_j, l_i, l_{i+1})\}$
12             if $A(\tau', s_n) < A_{s_n}^{(i+1)}$ then $\tau_{i+1} \leftarrow \tau'$;  $s_{\text{CH}}^{(i)} \leftarrow s_j$;  $A_{s_n}^{(i+1)} \leftarrow A(\tau', s_n)$
13 return $A(\tau_k, b)$
```

Let $n = \max\{|l_i \cap l_{i+1}|\}$. Given a line $l \in \mathcal{L}$, a station $s \in \mathcal{S}$ and a time $t_0 \in \mathbb{N}$, let $f$ be the time to find the earliest trip $\tau$ with $L(\tau) = l$ und $D(\tau, s) \geq t_0$ (this time depends on the concrete implementation of the timetable). It is easy to see that the running time of the above algorithm is bounded by $\mathcal{O}(kn(\log n + f))$.

## 4 Maximizing the Unexpected Similarity

**Computing the optimum journey for a fixed timetable** Given two stops $a, b \in \mathcal{S}$ and a departure time $t_0 \in \mathbb{N}$, we can already compute the earliest arrival of a journey from $a$ to $b$ starting at time $t_0$. From now on, we aim to compute the latest departure time at $a$ when the latest *arrival* time $t_A$ at $b$ is given. For this purpose we present an algorithm that sweeps backwards in time and uses the previous algorithm Earliest-Arrival. This sweepline algorithm will later be extended to count journeys (instead of computing a single one) and can be used for finding robust journeys, i.e. journeys that are likely to arrive on time.

The sweepline algorithm works as follows. We consider the trips departing at stop $a$ before time $t_A$, sorted in reverse chronological order. Everytime we find a trip $\tau$ of any line departing at some time $t_0$, we check whether there exists a route $r = \langle L(\tau), l_2, \dots, l_k \rangle \in \mathcal{R}$ that starts with the line $L(\tau)$. If yes, then we use the previous algorithm to compute the earliest arrival time at $b$ when we depart at $a$ at time $t_0$ and follow the route $r$. If the time computed is not later than $t_A$, we found the optimal solution and stop the algorithm. Otherwise we continue with the previous trip departing from $a$.

**Finding robust journeys** We will now describe how to compute robust journeys using the approach of Buhmann et al. [4]. We stress up front that this is "learning"-style algorithm and that it, in particular, does not specifically aims at optimizing some "robustness" criterion (such as the fraction of successes in the recorded timetables). Let $a, b \in \mathcal{S}$ be the departure and the target stop of the journey, $t_A$ be the latest arrival time at $b$, and $\mathcal{T}$ be a set of recorded timetables for comparable time periods (e.g., daily recordings for the past Mondays). For a timetable $T \in \mathcal{T}$ and a value $\gamma$, the *approximation set* $A_\gamma(T)$ contains a route $r \in \mathcal{R}$ iff there exists a journey along the route $r$ that starts at $a$ at time $t_A - \gamma$ or later and arrives at $b$ at time $t_A$ or earlier (both times refer to timetable $T$). The major advantage of this definition over classical approximation definitions (such as multiplicative approximation) is

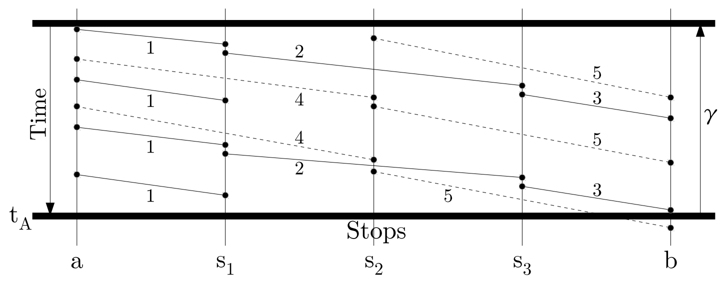


**Figure 3** An example with five lines $\{1,\dots,5\}$ and two routes $r_1 = \langle 1,2,3\rangle$ (solid) and $r_2 = \langle 4,5\rangle$ (dotted). The $x$-axis illustrates the stops $\{a, s_1, s_2, s_3, b\}$, whereas the $y$-axis the time. If a trip leaves a stop $s_d$ at time $t_d$ and arrives at a stop $s_a$ at time $t_a$, it is indicated by a line segment from $(s_d, t_d)$ to $(s_a, t_a)$. We have $\mu^T_\gamma(r_1) = 3$ and $\mu^T_\gamma(r_2) = 1$.

that we can consider multiple recorded timetables at the same time, and that the parameter $\gamma$ still has a direct interpretation as the time that we depart before $t_A$. Especially, if we consider approximation sets $A_\gamma(T_1),\dots,A_\gamma(T_k)$ for $T_1,\dots,T_k \in \mathcal{T}$, every set contains only routes that appear in the same time period and are therefore comparable among different approximation sets.

To identify *robust* routes when only two timetables $T_1, T_2 \in \mathcal{T}$ are given, we consider $A_\gamma(T_1) \cap A_\gamma(T_2)$: the only chance to find a route that is likely to be good in the future is a route that was good in the past for *both* recorded timetables. The parameter $\gamma$ determines the size of the intersection: if $\gamma$ is too small, the intersection will be empty. If $\gamma$ is too large, the intersection contains many (and maybe all) routes from $a$ to $b$, and not all of them will be a good choice. Assuming that we knew the optimal parameter $\gamma_{\mathrm{OPT}}$, we could pick a route from $A_{\gamma_{\mathrm{OPT}}}(T_1) \cap A_{\gamma_{\mathrm{OPT}}}(T_2)$ at random. Buhmann et al. [4] suggest to set $\gamma_{\mathrm{OPT}}$ to the value $\gamma$ that maximizes the so-called *similarity*

$$S_\gamma = \frac{|A_\gamma(T_1) \cap A_\gamma(T_2)|}{|A_\gamma(T_1)||A_\gamma(T_2)|}. \tag{2}$$

Notice that up to now we did not consider how often a route is realized by a journey in a recorded timetable. This is undesirable from a practical point of view: when we pick a route from $A_{\gamma_{\mathrm{OPT}}}(T_1) \cap A_{\gamma_{\mathrm{OPT}}}(T_2)$ at random, the probability to obtain a route should depend on how frequently it is realized. Therefore we change the definition of $A_\gamma(T)$ to a *multiset* of routes, and $A_\gamma(T)$ contains a route $r$ as often as it is realized by a journey starting at time $t_A - \gamma$ or later, and arriving at time $t_A$ or earlier. Figure 3 shows an example with five lines $\{1,\dots,5\}$ and two routes $r_1 = \langle 1,2,3\rangle$ and $r_2 = \langle 4,5\rangle$. We have $\mu^T_\gamma(r_1) = 3$: taking the second 1 and the second 2 (from above) as well as taking the third 1 and the second 2 are counted as different journeys since the departure times at $a$ differ. On the other hand, by our definition of journey we have to take the first occurence of a line that arrives, thus taking the first 1 and waiting for the second 2 is *not* counted.

Now the approximation set $A_\gamma(T)$ can be represented by a function $\mu^T_\gamma : \mathcal{R} \to \mathbb{N}_0$, where for a route $r \in \mathcal{R}$, $\mu^T_\gamma(r)$ is the number of journeys starting at time $t_A - \gamma$ or later, arriving at time $t_A$ or earlier and following the route $r$. Thus, we have $|A_\gamma(T)| = \sum_{r\in\mathcal{R}} \mu^T_\gamma(r)$, and for two recorded timetables $T_1, T_2$, we need to compute

$$\gamma_{\mathrm{OPT}} = \arg\max_\gamma \frac{\sum_{r\in\mathcal{R}} \min(\mu^{T_1}_\gamma(r), \mu^{T_2}_\gamma(r))}{\left(\sum_{r\in\mathcal{R}} \mu^{T_1}_\gamma(r)\right)\cdot\left(\sum_{r\in\mathcal{R}} \mu^{T_2}_\gamma(r)\right)}. \tag{3}$$

After computing the value $\gamma_{\mathrm{OPT}}$, we pick a route $r$ from $A_{\gamma_{\mathrm{OPT}}}(T_1) \cap A_{\gamma_{\mathrm{OPT}}}(T_2)$ at random according to the probability distribution defined by

$$p_r := \frac{\min(\mu^{T_1}_{\gamma_{\mathrm{OPT}}}(r), \mu^{T_2}_{\gamma_{\mathrm{OPT}}}(r))}{\sum_{r \in \mathcal{R}} \min(\mu^{T_1}_{\gamma_{\mathrm{OPT}}}(r), \mu^{T_2}_{\gamma_{\mathrm{OPT}}}(r))}, \tag{4}$$

and search in the planned timetable for a journey from $a$ to $b$ that departs at time $t_A - \gamma_{\mathrm{OPT}}$ or earlier, and that arrives at time $t_A$ or earlier.

**Computing the similarity** For $i \in \{1, 2\}$, we represent the function $\mu^{T_i}_{\gamma}$ by an $|\mathcal{R}|$-dimensional vector $\mu_i$ such that $\mu_i[r] = \mu^{T_i}_{\gamma}(r)$ for every $r \in \mathcal{R}$. We can compute the value $\gamma_{\mathrm{OPT}}$ by a simple extension of the aforementioned sweepline algorithm. The modified algorithm again starts at time $t_A$, and considers all trips in $T_1$ and $T_2$ in reverse chronological order. The sweepline stops at every time when one or more trips in $T_1$ or in $T_2$ depart. Assume that the sweepline stops at time $t_A - \gamma$, and assume that it stopped at time $t_A - \gamma' > t_A - \gamma$ in the previous step. Of course, we have $\mu^{T_i}_{\gamma}(r) \geq \mu^{T_i}_{\gamma'}(r)$ for every $r \in \mathcal{R}$ and $i \in \{1, 2\}$. Let $\tau_1, \ldots, \tau_k$ be the trips that depart in $T_1$ or $T_2$ at time $t_A - \gamma$. The idea is to compute the values of $\mu_i$ (representing $\mu^{T_i}_{\gamma}$) from the values computed in the previous step (representing $\mu^{T_i}_{\gamma'}$). This can be done as follows: for every trip $\tau_j$ occuring in $T_i$ and departing at time $t_A - \gamma$, we check whether there exists a route $r \in \mathcal{R}$ starting with $L(\tau_j)$. If yes, we distinguish two cases:

1. If $\mu_i[r] = 0$, then $\mu^{T_i}_{\gamma'}(r) = 0$, thus $r \notin A_{\gamma'}(T_i)$. If there exists a journey from $a$ to $b$ along $r$ departing at time $t_A - \gamma$ or later, and arriving at time $t_A$ or earlier, then $A_{\gamma}(T_i)$ contains $r$ exactly once. Thus, if EARLIEST-ARRIVAL$(a, b, t_A - \gamma, r) \leq t_A$, we set $\mu_i[r] \leftarrow 1$.
2. If $\mu_i[r] > 0$, then $\mu^{T_i}_{\gamma'}(r) > 0$, thus $A_{\gamma'}(T_i)$ contains $r$ at least once. Thus, there exists a journey from $a$ to $b$ along $r$ departing at time $t_A - \gamma'$ or later, and arriving at time $t_A$ or earlier. Since $\tau_i$ is the only possibility to depart at $a$ between time $t_A - \gamma$ and $t_A - \gamma'$, $\tau_i$ is the first trip on a journey we never found before. Therefore it is sufficient to simply increase $\mu_i[r]$ by 1.

Up to now, we did not define when the algorithm terminates. In fact we stop if $\gamma$ exceeds a value $\gamma_{\mathrm{MAX}}$. Let $t_A - \gamma_i$ be the starting time of an optimal journey in $T_i$. Of course, $\gamma_{\mathrm{MAX}}$ has to be larger than $\max\{\gamma_1, \gamma_2\}$. In our experimental evaluation, we set $\gamma_{\mathrm{MAX}}$ to be one hour before $t_A$; good choices for $\gamma_{\mathrm{MAX}}$ will be investigated in further experiments.

## 5 Journey Reliability

**Success rate as reliability** Having several recorded timetables at our disposal, and a journey from $a$ to $b$, a natural approach to assess its *reliability* with respect to the given latest arrival time $t_A$ is to check how many times in the past the journey finished before $t_A$. Normalized by the total number of recorded timetables, we call this success rate the *coupled reliability*. This is the least information about robustness one would wish to obtain from online routing services when being presented, upon a query to the system, with a set of routes from $a$ to $b$.

**Few recorded timetables** The generalizing expressiveness of coupled reliability is limited (and biased towards outliers in the samples) if the number of recorded timetables is small. If lines in the considered transportation network suffer from delays (mostly) independently, we can heuristically extract from each of the $m$ given recorded timetables $T_1, \ldots, T_m$ an individual timetable $T(i, l)$ for every line $l$ (storing just the travelled times of the specific line $l$ in timetable $T_i$), and then evaluate the considered journey on every relevant combination of these individual *decoupled* timetables. This enlarges the number of evaluations of the journey and thus has a chance to better generalize/express the observed travel times as typical situation.

**Decoupling the timetables** We can formally describe this process as follows. We consider $m$ recorded timetables $T_1, \dots, T_m$, and we consider a journey $J$ from stop $a$ to stop $b$, specified by a departure time $t_D$, by a sequence of lines $\langle l_1, \dots, l_k \rangle$, and by a sequence of transfer stops $\langle s^{(1)}_{\text{CH}}, \dots, s^{(k-1)}_{\text{CH}} \rangle$.

We say that journey $J$ is *realizable in* $\langle T(i_1, l_1), T(i_2, l_2), \dots, T(i_k, l_k) \rangle$, $i_1, \dots, i_k \in \{1, \dots, m\}$, *with respect to a given latest arrival time* $t_A$, if for every line $l_j$ there exists a trip $t_j$ (of the line $l_j$) in $T(i_j, l_j)$ such that

1. The departure time of trip $t_1$ from stop $a$ is after $t_D$,
2. the arrival time of trip $t_k$ at stop $b$ is before $t_A$, and
3. for every $j = 1, \dots, k-1$, the arrival time of trip $t_j$ at stop $s^{(j)}_{\text{CH}}$ is before the departure time of trip $t_{j+1}$ at the same stop.

**Decoupled reliability** Clearly, there are $m^k$ ways to create a $k$-tuple $\langle T(i_1, l_1), \dots, T(i_k, l_k) \rangle$. Let $M$ denote the number of those $k$-tuples in which journey $J$ is realizable with respect to a given $t_A$. We call the ratio $\frac{M}{m^k}$ the *decoupled reliability* of journey $J$ with respect to the latest arrival time $t_A$.

**Computational issues** Computing the coupled reliability is very easy: For every timetable $T_i \in \{T_1, \dots, T_m\}$ we need to check whether the journey in question finished before time $t_A$ or not. This can be done by a simple linear time algorithm that simply “simulates” the journey in the timetable $T_i$, and checks whether the arrival time of the journey lies before or after $t_A$. The computation of decoupled reliability is not so trivial anymore, as the straightforward approach would require to enumerate all $m^k$ $k$-tuples $\langle T(i_1, l_1), \dots, T(i_k, l_k) \rangle$, and thus an exponential time. In the following section, we present an algorithm that avoids such an exponential enumeration.

**Computing decoupled reliability** We can reduce the enumeration of all $k$-tuples $\langle T(i_1, l_1), T(i_2, l_2), \dots, T(i_k, l_k) \rangle$ by observing that the linear order of the lines in journey $J$ allows to use dynamic-programming. Let us denote for simplicity the boarding, transfer, and arrival stops of journey $J$ as $s_0, s_1, \dots, s_k$, where $s_0 = a$, $s_k = b$, and $s_j = s^{(j)}_{\text{CH}}$ for $j = 1, \dots, k-1$. For every stop $s_{j-1}$, $j = 1, \dots, k$, we store for every time event $t$ of a departing trip $\tau$ of line $l_j$ (in any of the timetables $T_1, \dots, T_m$) a “success rate” of the journey $J$: the fraction $SR[s_{j-1}, t]$ of all tuples $\langle T(i_j, l_j), \dots, T(i_k, l_k) \rangle$ in which the sub-journey of $J$ from $s_{j-1}$ to $s_k$ starting at time $t$ is realizable. For time $t$ not being a departure event, we extend the definition and set $SR[s_{j-1}, t] := SR[s_{j-1}, t']$, where $t'$ is the nearest time in the future for which a departing event exists. Having this information for every $j$, the decoupled reliability of $J$ is then simply $SR[s_0, t_D]$.

We can compute $SR[s_{j-1}, t]$ in the order of decreasing values of $j$. We initially set $SR[s_k, t_A] = 1$ (denoting that the fraction of successful sub-journeys arriving in $s_k$ is 1, if the sub-journey starts in $s_k$ and before $t_A$). The dynamic-programming like fashion for computing $SR[s_{j-1}, t]$ at any time $t$ then follows from the following recurrence:

$$SR[s_{j-1}, t] = \frac{1}{m} \sum_{i=1}^{m} SR[s_j, t_i], \tag{5}$$

where $t_i$ is the earliest arrival time of line $l_j$ at stop $s_j$ if the line uses timetable $T_i$ and does not depart before time $t$ from $s_{j-1}$.

When implementing the algorithm, we can save the (otherwise linear) time computation of the values of $t_i$ from the recurrence by simply storing this value and updating if needed.

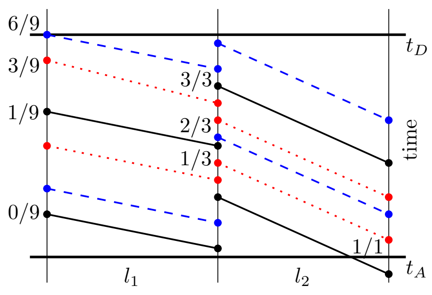


**Figure 4** A journey with two lines $l_1$ and $l_2$ and three timetables (solid black, dotted red, dashed blue). The fractions denote the stored values of $SR[s_j, t]$.

Figure 4 illustrates the algorithm, and the resulting decoupled reliability of 6/9. The running time of a naive implementation is $\mathcal{O}(k \cdot (m + e \log e))$, where $e$ is the maximum number of considered tram departing events at any station $s_j$.

## 6 Small Experimental Evaluation

In this section we describe and comment on a small experimental evaluation of the proposed approach to robust routing in public transportation networks. We first describe few observations/properties of our approach that serve as a kind of "mental" experiment. We have also implemented the proposed algorithms, and we report on our preliminary experiments with *real* public networks and *artificially* generated delays.

**Properties of the approach** Let $T_1$ and $T_2$ be two recorded timetables (from which we want to learn how to travel from stop $a$ to stop $b$ and arrive there before $t_A$). Consider the situation where the best journey $J$ to travel from $a$ to $b$ in timetable $T_1$ is the same as the best journey to travel from $a$ to $b$ in timetable $T_2$. Assuming that $T_1$ and $T_2$ represent typical delays, common sense dictates to use the very same journey $J$ also in the future. This is exactly what our approach does as well. Recall that $S_\gamma \leq 1$. Let $r$ be the route that corresponds to the journey $J$. In our case, setting $\gamma$ so that $A_\gamma(T_1) = A_\gamma(T_2) = \{r\}$, we get that $S_\gamma = \frac{|A_\gamma(T_1) \cap A_\gamma(T_2)|}{|A_\gamma(T_1)||A_\gamma(T_2)|} = 1$, and thus our approach computes the very same $\gamma$ and returns the journey $J$ as the recommendation to the user. These considerations can be generalized to the cases such as the one where $A_\gamma(T_1) = \{r\}$, $r \in A_\gamma(T_2)$, in which again $J$ will be returned as the recommendation to the user.

If only a reliable journey is required, and the travel time is not an issue, then suggesting to depart few days before $t_A$ is certainly sufficient. We now demonstrate that our approach does not work along these lines, and that it in fact reasonably balances the two goals *robustness* and *travel time*. We consider the symmetric situation where both $|A_\gamma(T_1)|$ and $|A_\gamma(T_2)|$ grow with $\gamma$ in the same way, i.e., for every $\gamma$, $|A_\gamma(T_1)| = |A_\gamma(T_2)|$. Let us only consider discrete values of $\gamma$, and let $\gamma_1$ be the largest $\gamma$ for which $A_{\gamma_1}(T_1) \cap A_{\gamma_1}(T_2) = \emptyset$. Let $x = |A_{\gamma_1}(T_1)|$. Then, for every $\gamma > \gamma_1$, $S_\gamma = \frac{\Delta_\gamma}{(x+\Delta_\gamma)^2}$ for some values of $\Delta_\gamma$. Simple calculation shows that $S_\gamma$ is maximized for $\Delta_\gamma = x$. We can interpret $x$ as the number of failed routes (that would otherwise make it if no delays appear). Then, $S_\gamma$ is maximized at the point that allows for another $\Delta_\gamma = x$ routes to joint the approximation sets $A_\gamma(T_i)$. Thus, the more disturbed the timetables are, the more "backward" in time we need to search for a robust route.

**Experimental evaluation** We implemented the algorithms presented in the sections 2, 3 and 4 in Java 7. The experiments were performed on one core of an Intel Core i5-3470 CPU

**Table 1** Comparison of the described methods over 100 test cases.

| | on time | less than 5 min late | less than 10 min late | avg arrival time | avg earlier depart. than Opt in $T_3$ |
|---|---|---|---|---|---|
| Unexpected Similarity, pick u.a.r. | 88% | 95% | 97% | 7:54 | 3.14 |
| Unexp. Sim., pick max. # occurences | 89% | 94% | 97% | 7:54 | 3.22 |
| Optimum in $T$ | 31% | 48% | 60% | 8:07 | -7.9 |
| 2nd Optimum in $T$ | 49% | 64% | 76% | 7:57 | 2.14 |
| Opt. in $T$ + end buffer time | 41% | 57% | 70% | 8:03 | -3.26 |
| Buffer time 3 min | 55% | 71% | 83% | 7:59 | 0.02 |
| Buffer time 5 min | 66% | 81% | 88% | 7:56 | 4.43 |

clocked at 3.2 GHz with 4 GB of RAM running Debian Linux 7.0. We used the combined tram and bus network of Zurich as input. It has 611 stops and 90 different line IDs. In our experiments, the actual number of lines itself is much higher (471), since multiple lines may operate under the same ID (e.g., lines in opposite directions, or lines coming from or returning to the depot). The planned timetable $T$ that we used is the official one for the Zurich network. However, trips departing before 6 a.m. or after 10 p.m. were ignored (since the timetable is only valid for 24 hours, trips starting before and ending after midnight are virtually interrupted at midnight, leading to a large number of lines).

We set the latest arrival time $t_A$ to 8 a.m., and carefully chose a small set of problematic stops $S'$ where delays usually occur. Then we generated 100 pairs of stops $(a, b)$ uniformly at random. For each pair, we generated three timetables $T_1$, $T_2$ and $T_3$ from $T$ by delaying every trip $\tau$ in $T$ between 0 and 3 minutes at every station $s \in S'$ (if $s$ occurs on $\tau$). These delays are 0 or 3 minutes with probability 1/8, and 1 or 2 minutes with probability 3/8. $T_1$ and $T_2$ are used as input to the algorithm, and the arrival time of the computed journey is measured in $T_3$. We use the following methods for computing the journey.

1. **Maximizing the Unexpected Similarity** Compute a route using the approach described in section 4. We consider two ways to pick a route from the intersection: 1) choose uniformly at random; 2) Choose the one with the maximum number of occurrences.
2. **Optimum in $T$** Find the best or the second best journey according to the planned timetable $T$. Compute also the latest journey arriving in $T$ five minutes before $t_A$.
3. **Buffer time for transfers** Consider the latest journey from $a$ to $b$ that arrives on time in $T$ such that at each transfer stop it have to wait for an additional "buffer time". We experiment with buffer times of 1 – 5 minutes.

For each of these statistics, we computed the following numbers (see Table 1): Percentage of the experiments where the proposed journey arrives on time, how often it arrives at most 5 minutes late, and how often it arrives at most 10 minutes late. We also computed the average arrival time of the journeys proposed by each method as well as the average difference between the departure time of the proposed journey to the optimal journey in $T_3$.

The average time for computing the optimum solution is 127ms, the time to compute a robust journey by using Unexpected Similarity is 262ms. We observed that our algorithm produces journeys that are on time in high percentage of cases, and on average we propose to depart only around 3 minutes earlier than the optimum in $T_3$, thus the cost we pay for this robustness is quite low. In comparison, the other considered approaches achieve much lower success rates. Even the generous buffer time of 5 minutes turns out not to be enough to beat our approach, which is rather surprising given the small delays in the considered timetables.

## 7 Discussion

We presented a novel framework for robust routing in frequent and dense urban public transportation networks based on observations of past traffic data. We introduced a new concept to describe a travel plan, a *journey*, that is not only well suited for our robustness issues, but also represents a natural and convenient description for the traveler. We also provided a bag of algorithmic tools to handle this concept, tailored towards the proposed robustness measures. We described a simple way to assess the reliability of a given journey. We also used a different approach to robustness and described how to find a robust journey according to it. We are preparing further experiments to confirm efficiency of the presented algorithms and to evaluate the quality of the computed robust journeys.

Future work is to examine how the described methods can be extended to support a fully multi-modal scenario, e.g., how to integrate walking. We believe that the modelling itself is easy, while the performance of the algorithms will decrease significantly unless we develop special techniques. Also considering and exploring different robustness concepts for journeys may be worthwhile.

## References

1 Hannah Bast, Erik Carlsson, Arno Eigenwillig, Robert Geisberger, Chris Harrelson, Veselin Raychev, and Fabien Viger. Fast routing in very large public transportation networks using transfer patterns. In Mark Berg and Ulrich Meyer, editors, *Algorithms – ESA 2010*, volume 6346 of *LNCS*, pages 290–301. Springer Berlin Heidelberg, 2010.

2 Reinhard Bauer, Daniel Delling, and Dorothea Wagner. Experimental study of speed up techniques for timetable information systems. *Networks*, 57(1):38–52, 2011.

3 Justin Boyan and Michael Mitzenmacher. Improved results for route planning in stochastic transportation. In *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 895–902. Society for Industrial and Applied Mathematics, 2001.

4 Joachim M. Buhmann, Matúš Mihalák, Rastislav Šrámek, and Peter Widmayer. Robust optimization in the presence of uncertainty. In Robert D. Kleinberg, editor, *ITCS*, pages 505–514. ACM, 2013.

5 Daniel Delling, Thomas Pajor, and Dorothea Wagner. Engineering time-expanded graphs for faster timetable information. In Ravindra K. Ahuja, Rolf H. Möhring, and Christos D. Zaroliagis, editors, *Robust and Online Large-Scale Optimization*, volume 5868 of *LNCS*, pages 182–206. Springer Berlin Heidelberg, 2009.

6 Daniel Delling, Thomas Pajor, and Renato F Werneck. Round-based public transit routing. *Algorithm Engineering and Experiments (ALENEX)*, pages 130–140, 2012.

7 Julian Dibbelt, Thomas Pajor, Ben Strasser, and Dorothea Wagner. Intriguingly simple and fast transit routing. In Vincenzo Bonifaci, Camil Demetrescu, and Alberto Marchetti-Spaccamela, editors, *SEA*, volume 7933 of *LNCS*, pages 43–54. Springer, 2013.

8 Yann Disser, Matthias Müller-Hannemann, and Mathias Schnee. Multi-criteria shortest paths in time-dependent train networks. In *Experimental Algorithms*, pages 347–361. Springer, 2008.

9 H Frank. Shortest paths in probabilistic graphs. *Operations Research*, 17(4):583–599, 1969.

10 Marc Goerigk, Martin Knoth, Matthias Müller-Hannemann, Marie Schmidt, and Anita Schöbel. The price of robustness in timetable information. In *11th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems*, pages 76–87, 2011.

11 Matthias Müller-Hannemann and Mathias Schnee. Efficient timetable information in the presence of delays. In Ravindra K. Ahuja, Rolf H. Möhring, and Christos D. Zaroliagis,

editors, *Robust and Online Large-Scale Optimization*, volume 5868 of *LNCS*, pages 249–272. Springer, 2009.

12 Matthias Müller-Hannemann, Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. Timetable information: Models and algorithms. In *Algorithmic Methods for Railway Optimization*, pages 67–90. Springer, 2007.

13 Evdokia Nikolova, Jonathan A Kelner, Matthew Brand, and Michael Mitzenmacher. Stochastic shortest paths via quasi-convex maximization. In *Algorithms–ESA 2006*, pages 552–563. Springer, 2006.

14 Evangelia Pyrga, Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. Efficient models for timetable information in public transportation systems. *Journal of Experimental Algorithmics (JEA)*, 12:2–4, 2008.

# Delay-Robustness of Transfer Patterns in Public Transportation Route Planning *

**Hannah Bast, Jonas Sternisko, and Sabine Storandt**

**Albert-Ludwigs-Universität Freiburg**
**Freiburg, Germany**
{bast,sternis,storandt}@informatik.uni-freiburg.de

## Abstract

Transfer pattern routing is a state-of-the-art speed-up technique for finding optimal paths which minimize multiple cost criteria in public transportation networks. It precomputes sequences of transfer stations along optimal paths. At query time, the optimal paths are searched among the stored transfer patterns, which allows for very fast response times even on very large networks. On the other hand, even a minor change to the timetables may affect many optimal paths, so that, in principle, a new computation of all optimal transfer patterns becomes necessary. In this paper, we examine the robustness of transfer pattern routing towards delay, which is the most common source of such updates. The intuition is that the deviating paths caused by typical updates are already covered by original transfer patterns. We perform experiments which show that the transfer patterns are remarkably robust even to large and many delays, which underlines the applicability and reliability of transfer pattern routing in realistic routing applications.

**1998 ACM Subject Classification** F.2.2 Nonnumerical Algorithms and Problems

**Keywords and phrases** Route planning, public transportation, transfer patterns, delay, robustness

**Digital Object Identifier** 10.4230/OASIcs.ATMOS.2013.42

## 1 Introduction

When traveling with public transportation, not only the absolute time of travel matters: Also the number of transfers and the total fare are important or, for instance, the reliability of connections along the journey. The goal of public transportation route planning is to find paths that minimize a multi-criteria cost function. Public transportation data is typically available as a set of timetables and can be modeled as a directed graph. Classical route planning algorithms perform a multi-criteria variant of Dijkstra's algorithm on the graph. To our knowledge, *transfer pattern routing* [1] is the fastest speed-up technique for this problem. After precomputing sequences of transfers along all optimal paths which uses quadratic time in the number of stations, it allows to find the Pareto-optimal paths in huge networks within a few milliseconds. Because of its excellent scalability, the idea of transfer pattern routing is employed by Google Maps. If the underlying network (read: the information of the timetables) changes, the precomputed transfer patterns become outdated and optimal results cannot be guaranteed anymore. Incorporating an update into the transfer patterns is hard, because the dependency between a changed connection and the affected transfer patterns is unclear. In principle, the whole expensive precomputation has to be done again. But this is impossible as in realistic settings there are often updates. Our idea is, that provided there

* Partially supported by a Google Focused Research Award.

are only minor changes to the network, the original transfer patterns are sufficient to find optimal routes in most cases.

Criticism of transfer pattern routing often refers to its theoretical suboptimality and lacking support for dynamic scenarios. At the time of writing, there are no publications about if and how real-time updates can be handled by transfer pattern routing. But this is an important aspect for the practicability of the algorithm, because route planning service providers wish to recompute the transfer patterns only occasionally, when long-term changes to the timetables are made. The main contribution of this work resides in empirically proving the reliability of transfer pattern routing for location-to-location queries in the context of real-time updates. We evaluate the quality of transfer patterns in different global delay scenarios and study the immediate effect of delaying connections involved in optimal paths. Moreover, we investigate on which parameters the robustness depends and how it can be increased.

## 2 Related Work

Transfer pattern routing has been introduced by Bast et al. [1]. The authors outline the key components of the algorithm and present a set of techniques and heuristics to render the computation of transfer patterns feasible. Most notably is the concept of computing only parts of transfer patterns up to important stations and combining these parts at query time. Geisberger [6] elaborates how to compute transfer patterns in fully realistic settings with walking between stations and for answering location-to-location queries.

The requirements for a route planning algorithm in a dynamic network are analyzed in [10]. There have not been any publications on transfer patterns in such a setting yet. Speed-up techniques without a time-consuming precomputation and thus suitable for dynamic scenarios are for example SUBITO [3] and RAPTOR [4]. These approaches allow to find Pareto-optimal paths between stations in short time (SUBITO about 100 ms on German railway network, RAPTOR about 100 ms for London transit). However, the query times of these approaches cannot compete with transfer pattern routing on large networks (43 ms for North America).

A related field of research focuses on robustness to delay in the sense that the probability of missing a connection along a route is minimized. Most recently, a framework of algorithms based on a technique called *Connection Scan* has been introduced [5]. The authors report convincing average query times for finding the route with earliest arrival time (1.8 ms) and for multi-criteria profile queries (255 ms) on the London data set. However, it remains unclear how fast the algorithm answers one-to-one queries when more than one cost-criterion is minimized. Besides the solution of classical route planning problems, the authors apply Connection Scan to find alternative routes by minimizing the *expected* arrival time. Goerigk et al. [7] compute routes which are robust to delays in the sense that all transfers along a route are guaranteed, given a specific delay scenario. They observe that *strictly robust routes* last longer than the fastest routes, whereas *light robust routes* have a relatively small overhead. Keyhani et al. [9] introduce a stochastic model which rates the reliability of transfers along routes. Other than in those articles, robustness does not refer to routes with reliable transfers in this paper, but to *sustaining optimality*. Section 4.1 refers to the delay models of the aforementioned works in more detail.

## 3 Preliminaries

This section defines preliminary concepts and models. It explains the idea and components of transfer pattern routing.

### 3.1 Modeling timetables

A transit network is described by a set of timetables. It comprises information about stations $S$ (e.g. train stations or bus stops) and trips of transport vehicles. A *trip* $T$ serves a sequence of stations $\mathit{stops}(T) = (s_1, s_2, \ldots, s_n)$, $s_i \in S$ at arrival and departure times $(t_1^{arr}, t_1^{dep}), (t_2^{arr}, t_2^{dep}), \ldots (t_n^{arr}, t_n^{dep})$. Let $\mathit{stop}(T, s)$ denote the index of $s$ in the station sequence of $T$. We say a trip connects two stations $s_a$ and $s_b$, if $s_a, s_b \in \mathit{stops}(T)$ and $\mathit{stop}(T, s_a) < \mathit{stop}(T, s_b)$, and call $s_a$ and $s_b$ the start- and endpoint of the connection, respectively. Multiple trips which share the same sequence of stations and do not overtake each other form a *line*.

A *route* between two stations is a sequence of alternating rides on board of a vehicle and transfers between connections. The start- and endpoints of the $n$ connections along a route form a sequence of $2n$ stations, which is called *transfer pattern* of the route. For queries between two *locations* (not stations) $X$ and $Y$, there is an additional walking part at the beginning and the end. A routing algorithm answers a query with a set of routes that minimize multiple cost criteria. The *costs* of a route are the sum of the costs of all its connections and transfers. When comparing cost tuples, we say that $a$ dominates $b$ ($a < b$), if it is as good as $b$ in every component and better in at least one. $a$ and $b$ are incomparable, if neither $a < b$ nor $b < a$. The costs of optimal paths to a query are pairwise incomparable, they form a Pareto-set.

**Time-expanded Graph** We use publicly available timetable data following the General Transit Feed Specification (GTFS) format and model it as time-expanded graph according to Pyrga et al. [11]. Each departure and arrival event along a trip is explicitly modeled as a node with a timestamp. The successive nodes are connected with arcs of costs corresponding to the time difference between the two events. Beside *arrival* and *departure nodes*, there are nodes modeling transfers and waiting at a station. For each departure node, there is a *transfer node* with the same timestamp and an arc connecting it to the departure node. At each station, every transfer node is connected to the next transfer node in time. To model transfers between vehicles, an arc connects each arrival node to a subsequent transfer node. Usually, a traveler cannot instantly change from one vehicle to another. We model this with the difference between the arrival and the connected transfer node not being less than a fixed *transfer buffer* of 120 seconds. We want to find routes which minimize the time of travel and the number of transfers. Therefore, the arcs have a tuple weight consisting of the time difference between the connected nodes and the *penalty*, which is 1 for arcs from arrival to transfer nodes and 0 for all other arcs. In order to decrease the size of the graph, we remove departure nodes by redirecting incoming arcs to the respective successors.

**Walking Between Stations** In a model for realistic route planning, transfers involving walking between two stations must be possible. Therefore, we maintain an additional walking graph with arcs between neighboring stations and the duration of walking as costs. For the sake of simplicity, we take the straight-line distance between the connected stations and assume a fixed speed of 5 km/h to compute the costs. During search, when expanding a label at an arrival node at station $S$ and time $t_{arr}$, the walking graph is used to determine the first transfer node after $t_{arr} + walk(S,T) + transfer\ buffer$ for every neighbor station $T$. The time difference between the two nodes and a penalty of 1 are used as weight for the relaxed virtual arc. By restricting walking to happen between arrival and transfer nodes, this model implicitly forbids *via-walking* over multiple stations in a row.

## 3.2 Routing with Transfer Patterns

The *transfer pattern* of a route is the sequence of stations where a change of the transportation vehicle occurs, including the departure and arrival station. When considering all possible departure times at a station $A$, the optimal paths for journeys $A \to B$ form a set of transfer patterns. In public transportation, this set has typically only a few elements. For example, when traveling from Paris to Nice there is a direct TGV which leaves every other hour. In between its departure times, the journey with the earliest arrival time at Nice is one of two connections with transfers at Lyon or Marseilles. We say that *Paris – Nice*, *Paris – Lyon – Nice* and *Paris – Marseilles – Nice* are the optimal transfer patterns for this station pair.

The key idea of the algorithm is that the set of optimal transfer patterns between two stations $A$ and $B$ *at all times* form a search space, which is orders of magnitude smaller than the original graph. Given they are known, the Pareto-optimal paths at a specific time can be found among them. In short, the algorithm determines the optimal transfer patterns for all pairs of stations and searches on the graph described by the patterns.

**Computation of Optimal Transfer Patterns** Conceptually, the optimal transfer patterns for a station pair $A, B$ can be determined by running a multi-criteria variant of Dijkstra's algorithm from $A$. On the time-expanded graph[1], we compute the transfer patterns as described in [1]: From every station, a profile query determines the optimal paths to all reachable destinations. For every destination and its arrival nodes at times $t_1 < t_2 < \dots$ the *arrival-chain* algorithm selects a dominating subset among the set of labels consisting of (i) labels settled at $t_i$, and (2) labels settled at $t_{i-1}$ with duration increased by $t_i - t_{i-1}$. Every selected label corresponds to an optimal path, which is backtracked to its origin while recording stations where transfers happened. The resulting optimal transfer patterns are stored as a *directed acyclic graph (DAG)*. In extension to [1], we exploit the fact that the departure and destination station of a transfer pattern is always known from context [12]. This allows to store all patterns in one *joint DAG*, which automatically resolves redundancies and reduces the size of the data.

The precomputation has quadratic time effort in the number of stations. When computing transfer patterns for location-to-location queries (which we do), the arrival-chain algorithm has to consider all arrival events in the walking neighborhood $\mathcal{N}(s)$ of each destination $s$. With an unbound neighborhood radius the running time would become cubic. Therefore we limit walking to stations within 1 000 meters. Nevertheless, the precomputation is very expensive. This is the price for the very fast query times. In order to reduce its duration, we employ the concept of important stations (hubs) and compute only parts of transfer patterns [1]. Global (unlimited) transfer patterns are computed only from important stations. From all other stations, we compute the transfer patterns up to the first transfer at a hub and a maximum of three trips. Although this heuristic leads to a loss of optimality (the search cannot find optimal paths with more than two transfers, none of which is at a hub), in

[1] Note that our results are independent of the used graph model.

practice, only very few optimal paths are affected [1]. In Section 5 we will see its marginal effect on the optimality of the algorithm's results.

**Routing with Transfer Patterns** Once computed, the patterns between stations $A$ and $B$ describe a compact graph. A location-to-location query $X@t \to Y$ is answered by constructing a *query graph* and performing a search on it. The query graph is created from the transfer patterns between departure stations $s \in \mathcal{N}(X)$, the important stations and the destination stations $s' \in \mathcal{N}(Y)$ as demonstrated in [1, 6]. We follow the refinements of Geisberger [6] and distinguish between two nodes representing alternating arrival and departure events for each station. The arcs in this graph correspond to (walking-) transfers or direct connections between stations. During the search, the travel time along these arcs can be determined using an efficient data structure. We proceed in analogy to [1] and store trips grouped by lines like this:

| $\text{line}_{17}$ | $s_{14}$ | $s_9$ | $s_{56}$ | ... |
|---|---|---|---|---|
| $\text{trip}_1$ | 8:05 | 9:00 9:15 | 10:00 10:05 | ... |
| $\text{trip}_2$ | 8:35 | 9:30 9:45 | 10:30 10:35 | ... |
| ... | ... | ... | ... | ... |

For each station, we compute a list of incident lines with the respective position of the station along the line. For instance: $s_{14} : \{(\text{line}_{17}, 0), (\text{line}_{26}, 8), \ldots\}$, $s_{56} : \{(\text{line}_{12}, 6), (\text{line}_{17}, 2), \ldots\}$ and so on. To determine the next direct connection between two stations, their incidence-lists are intersected and the next trip of a line connecting both stations is determined. With the query graph consisting of only several hundred arcs and the direct connection queries taking 2-10$\mu s$ each, the total search time is only a few milliseconds.

## 4 Delay and Robustness

This section presents our delay model, points out the problem of frequent updates for a preprocessing-based algorithm and introduces our approach to handle delay with transfer pattern routing.

### 4.1 Delay Scenarios

Among different sources of real-time updates to timetables (trip cancellation, redirection, auxiliary connections, ...) we focus on the most common one, which is delay of trips. The literature distinguishes between primary delay (e.g. a train is late due to engine issues) and secondary delay (other trains waiting for the former) [10]. Delay models in related projects range from simplistic independence assumptions [5] over models which allow for delay to accrue [7] to sophisticated models respecting primary delay of trains, knock-on delay to other trips and delay due to waiting for late connections [8]. In a survey of stochastic models for delay, Yuan [13] successfully anneals distributions of non-negative primary delay with exponential functions. Once a trip is delayed, the propagation over successive connections follows complex rules. Refer to Berger et al. [2] for an overview and a stochastic model for delay propagation in timetables.

For the sake of simplicity and because data about real-time updates is hardly available, we focus on primary delay, ignore knock-on effects as well as scheduled security headways between trains and model delay independently between trips. In six different scenarios (Table 1), the set of trips is partitioned into groups of common average delay $\mathbb{E}\delta$. For each group, a random subset of all trips is selected. Every selected trip is delayed with time $\delta$ drawn from

**Table 1** How many trips are delayed by how much in our six delay scenarios.

| Scenario | Average delay $\mathbb{E}\delta$ | | |
|---|---|---|---|
| | 5 min | 15 min | 50 min |
| Low | 25% | - | - |
| Medium | - | 25% | - |
| High | - | - | 25% |
| Mix Low | 10% | 3% | 1% |
| Mix Normal | 20% | 10% | 5% |
| Mix Chaos | 40% | 40% | 20% |

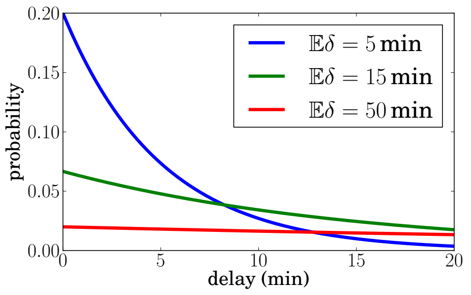


**Figure 1** Probability density functions for exponential distributions with mean $\mathbb{E}\delta$.

an exponential distribution with probability density function $pdf(\delta) = 1/\mathbb{E}\delta \cdot \exp(-1/\mathbb{E}\delta \cdot \delta)$ (Figure 1). The delay is inserted starting at a uniformly random stop $i$, i.e. the trip's times $(t_j^{arr}, t_j^{dep})$ are replaced with $(t_j^{arr} + \delta, t_j^{dep} + \delta)$ for $j \geq i$. We choose three different scenarios where one quarter of the connections are delayed with 5 (Low), 15 (Medium) an 50 minutes (High) in average. In addition, we generate three combined scenarios with an increasing mixture of average delay (Mix Low, Mix Normal and Mix Chaos). In the last scenario, every trip is delayed.

One might argue that this model is too far from reality. However, by modeling delay independently between trips, the scenarios become harder to deal with than in reality. If delay occurs frequently along a specific line or in a street prone to congestion, alternative routes are more obvious and could be retrieved during the precomputation. Furthermore, in typical metropolitan networks with high service frequencies, connections typically do not wait for delayed trips. Waiting policies in hierarchical train networks are designed such that the important transfers between trips are maintained and the resulting delay for waiting trips can be compensated during the remainder of their trip and knock-on delay to further connections is minimized. Therefore, we believe that in a refined model with realistic waiting rules transfer patterns will perform more robust than in our simplified model. We are working on another set of experiments with such a model, but by the time of writing there are no results yet.

## 4.2 Delay and Transfer Patterns

The routing algorithm finds optimal routes only if the precomputed transfer patterns are optimal. If a trip is delayed, it is possible that an optimal route previously taking this trip will resort to another connection, thereby changing its transfer pattern. Unfortunately, not only routes along the delayed trip are affected and it is hard to decide which transfer patterns have to be updated. To make this clear, think of a train which is delayed and stops at some station at 10:15 instead of 9:45. Another train arrives at the same station at 10:00. Passengers of this train may benefit from the delayed train and arrive at their destination earlier than with a regular connection.

Thus, the optimal transfer patterns have to be computed from scratch. But this is time-consuming: for example, the transfer pattern computation for New York requires around 800 core hours ([1], Table 3). Given a steady flow of updates to the timetables, it is impossible to keep the transfer patterns up to date. On the other hand, the data structure for direct connection lookup can be computed within a few minutes ([1], Table 2). Updating a single trip is fast, as we will prove. It is thus adaptable to frequent changes of the timetables. Our approach to deal with real-time updates is to update only the direct connection data, and search on query graphs generated from the original transfer patterns.

## 4.3 Updating the Direct Connection Data

Now we explain how a single trip in the direct connection data structure can be updated in real-time. When constructing the data from a collection of trips, we create a mapping from sequences of station ids to all lines that serve these stations in the given order. When updating a specific trip, the trip's stop times are changed within its line. If the line still has the FIFO-property (the trip does not overtake another trip and is not overtaken), we are done. Otherwise, the trip is removed from the line. If it does not fit into another line serving the same sequence of stations, a new line is created. The line id and the respective stop position along the line is added to the incidence list of every served station (see Section 3.2).

▶ **Lemma 1.** *Let $L$ denote the set of lines and let $trips(l)$ denote the trips of $l \in L$. Further, $C \subseteq L$ is the set of lines which share the same station sequence as $l$. Then updating a trip $T \in trips(l)$ in the direct connection data structure has running time*

$$O\left(\log|L| \cdot |stops(T)| + |C| \cdot (|stops(T)| + \log|trips(l)|)\right)$$

**Proof.** (1) Finding the trip in the line can be done in $O(\log|trips(l)|)$, because the trips of $l$ are sorted by time (Section 3.2). (2) Updating the trip's stop times is in $O(|stops(T)|)$. (3) Checking the FIFO-property takes $O(|stops(T)|)$. In case it is violated, candidate lines $C$ with the same same sequence of stations have to be found in $L$. This takes time $O(\log|L|\cdot|stops(T)|)$ using the mapping described above. Steps (1) and (3) have to be repeated for every $c \in C$ in the worst case. ◀

For example in New York City, there are 16 454 lines and $C$ has a maximum size of 66. For the most frequent lines, $|trips(l)|$ is 299 and for the longest trips $|stops(T)|$ is 117. Updating a trip takes 40–80 $\mu s$. Typical update rates are about 70 updates per second (German railway; primary delay, secondary delay and forecast) [10], so our approach clearly allows for real-time updates.

## 5 Experiments

In the previous section, we proposed to deal with delays by searching routes using the original transfer patterns (computed for the graph without delays) and updated direct connection data. This potentially leads to non-optimal responses. In this section, we present several experiments which show that this is rarely the case, even for many and large delays.

## 5.1 Global Delay Scenarios

**Method** We present experiments conducted on the data sets of Toronto (10 883 stations, 1.5 M departures) and New York City (16 765 stations, 2.3 M departures). The data can be accessed at `http://ad.informatik.uni-freiburg.de/publications`. The time-expanded graph is generated for a random weekday (from 1:00 am until 6:30 am the next day) and the transfer patterns are computed on this graph according to Section 3.2. This forms the baseline NULL. We apply different delay scenarios from Section 4.1 to the data sets and compute the direct connection data and the updated time-expanded graph from it. At search time, the query graphs are constructed from the transfer patterns computed on the original graph.

**Table 2** Classification of paths under different scenarios. Abbreviates ALMOST OPTIMAL $\sigma$ as $\sigma$.

**(a)** Toronto

| | OPTIMAL | A | B | BAD |
|---|---|---|---|---|
| NULL | 99.97% | 0.02% | 0.01% | 0.00% |
| LOW | 99.71% | 0.15% | 0.04% | 0.10% |
| MEDIUM | 99.55% | 0.22% | 0.06% | 0.17% |
| HIGH | 99.42% | 0.29% | 0.08% | 0.21% |
| MIX LOW | 99.81% | 0.10% | 0.03% | 0.06% |
| MIX NORMAL | 99.52% | 0.26% | 0.06% | 0.16% |
| MIX CHAOS | 97.46% | 1.40% | 0.34% | 0.80% |

**(b)** New York City

| | OPTIMAL | A | B | BAD |
|---|---|---|---|---|
| NULL | 99.99% | 0.01% | 0.00% | 0.00% |
| LOW | 99.87% | 0.01% | 0.00% | 0.04% |
| MEDIUM | 99.68% | 0.19% | 0.03% | 0.10% |
| HIGH | 99.72% | 0.17% | 0.02% | 0.09% |
| MIX LOW | 99.93% | 0.05% | 0.00% | 0.02% |
| MIX NORMAL | 99.89% | 0.07% | 0.01% | 0.03% |
| MIX CHAOS | 99.19% | 0.57% | 0.07% | 0.17% |

We generate random queries $X@t \to Y$ in the following manner: The departure and destination locations $X$ and $Y$ are drawn from the set of locations of the stations $S$, taking into account the number of departing connections $n_s$ at each station $s \in S$: The probability of selecting $s$ is $p_s = \sqrt{n_s}/\sum_{s' \in S}\sqrt{n_{s'}}$. To avoid trivial connections, the two locations must be more than 2 000 meters away from each other. The departure time $t$ is drawn from an interval of 24 hours starting at 4:00 am. To account for varying traffic density during the day, departure times during the rush hours are selected twice as often.

The random queries are answered by transfer pattern routing. The resulting paths are compared to reference routes. In order to compute the latter, the delayed time-expanded graph is extended with two nodes $x, y$ representing the source and target location $X$ and $Y$. For each station $s \in \mathcal{N}(X)$, $x$ is connected to the first transfer node of $s$ after time $t + walk(X, s)$ and every arrival node at $s' \in \mathcal{N}(Y)$ is connected to $y$ by an arc of costs $walk(s', Y)$. On this extended graph, a multi-criteria Dijkstra is used to determine the optimal paths.

In this setting, we evaluate the robustness in each scenario. For each response to a query, the paths found by transfer pattern routing are classified independently as follows: If a path of equal costs is among the reference paths, the response is OPTIMAL. For every Dijkstra-generated path which has no correspondent of equal costs, the most similar path in terms of penalty is selected. If there is a path with the same penalty, the duration difference is inspected. If the path found by transfer pattern routing is less than 5% of the total travel time *and* less than five minutes slower than the reference path, it is ALMOST OPTIMAL A. If it is not classified as ALMOST OPTIMAL A but less than 10% *and* less than ten minutes late, it is ALMOST OPTIMAL B. Otherwise, the path is classified as BAD.

**Results** Tables 2a and 2b show the results of our experiments with 50 000 random queries with at least one feasible route found by the reference algorithm. For each delay scenario, the found paths were classified (about 82 000 paths in Toronto, 67 000 in New York City). The classification results show the influence of the scenarios: With increasing average delay, the share of optimal paths decreases. Although we are using the important station heuristic, almost all paths found by transfer pattern routing are optimal for the baseline NULL. The few suboptimal paths are dominated by paths with more than two transfers without an important station, which cannot be found because of the restriction of local profile queries to three trips (see Section 3.2).

Among the results for the different scenarios there are just very few suboptimal paths. Even for the worst scenario the share of suboptimal paths is below 2.6% for Toronto, and

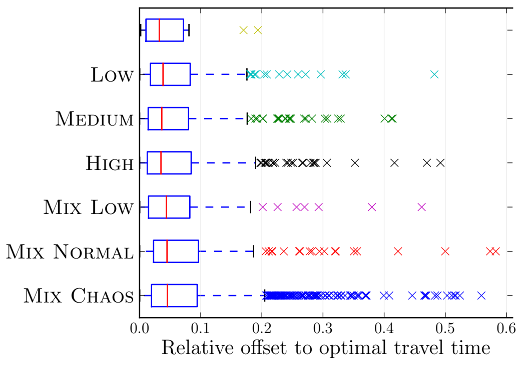
**Table 3** Suboptimal paths: Relative offset to optimal travel time. Summary statistics of distributions for different delay scenarios. For example, in Toronto under the scenario LOW, 25% of the suboptimal paths are at most 0.02 times slower than the optimal path.

**(a)** Toronto

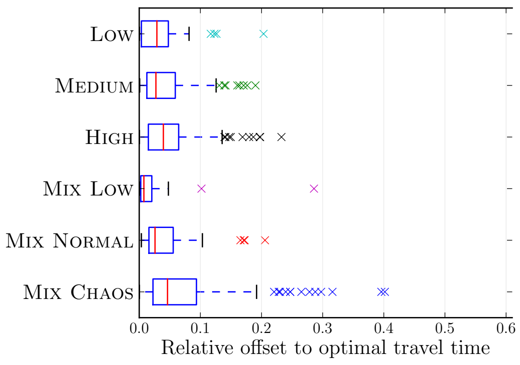
| | N | $Q_{0.25}$ | $Q_{0.5}$ | $Q_{0.75}$ | max |
|---|---|---|---|---|---|
| LOW | 276 | 0.02 | 0.04 | 0.08 | 4.95 |
| MEDIUM | 441 | 0.01 | 0.04 | 0.08 | 3.96 |
| HIGH | 552 | 0.01 | 0.04 | 0.08 | 3.17 |
| MIX LOW | 184 | 0.01 | 0.04 | 0.08 | 0.74 |
| MIX NORMAL | 451 | 0.02 | 0.04 | 0.10 | 0.69 |
| MIX CHAOS | 2300 | 0.02 | 0.05 | 0.09 | 4.42 |

**(b)** New York City

| | N | $Q_{0.25}$ | $Q_{0.5}$ | $Q_{0.75}$ | max |
|---|---|---|---|---|---|
| LOW | 46 | 0.00 | 0.03 | 0.05 | 0.20 |
| MEDIUM | 149 | 0.01 | 0.03 | 0.06 | 0.19 |
| HIGH | 144 | 0.01 | 0.04 | 0.06 | 0.23 |
| MIX LOW | 24 | 0.00 | 0.01 | 0.02 | 0.29 |
| MIX NORMAL | 41 | 0.02 | 0.03 | 0.06 | 0.21 |
| MIX CHAOS | 332 | 0.02 | 0.05 | 0.09 | 0.40 |



**(a)** Toronto



**(b)** New York City

**Figure 2** Suboptimal paths: Boxplot for relative time of travel compared to the optimal path with equal number of transfers. The red line marks the median, the box contains the interval between the upper and lower quartile of the data. The whiskers have a length of 1.5 times the distance between the quartiles. The crosses mark outliers. Outliers above 0.6 are not shown.

below 1% for New York City. Beside this, most of the suboptimal paths are quite close to the optimum: The major part is classified as ALMOST OPTIMAL A and the share of BAD paths is never larger than 0.8%. For suboptimal paths, Tables 3a, 3b and Figures 2a, 2b show the distribution of the relative differences to the corresponding optimal path. The influence of the scenarios' average delay reflects similarly in the distributions as for the classification results above. We observe that the median of the distributions is below 0.05 for both data sets in all scenarios. There are a few outliers, some of which are much worse than the optimal path (at most factor 4.95 for Toronto, factor 0.37 for New York City). Manual inspection of these critical outliers showed that they typically stem from queries between remote locations with bad connectivity. For example, the worst route in Toronto misses the last connection before midnight and has to wait for six hours. Note that only paths which are dominated by a reference path of equal number of transfers are reported here. Therefore, the number of paths in the Tables 3a, 3b is slightly smaller than the number of suboptimal paths in the classification tables.

In summary, the results indicate that transfer patterns are very robust to delay. Even in the worst scenario the share of suboptimal responses is very small, and most of these paths are almost optimal. Furthermore, the results show that the limit of three trips for local transfer patterns leads only to very few suboptimal results.

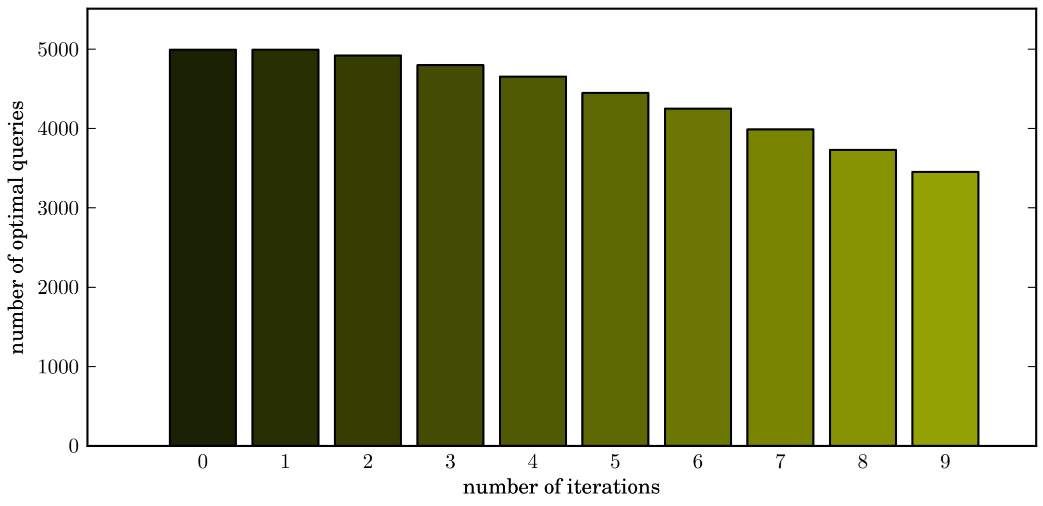


**Figure 3** Number of queries which are still answered optimally after iteratively adding systematic delay of 30 minutes to the optimal paths and repeating the query. Results for 5 000 queries with a limit of nine iterations on New York City.

## 5.2 Controlled Delay

In our opinion, the global scenarios discussed beforehand model real transportation networks quite sufficiently. On the other hand, delaying random trips in a memoryless fashion does not clearly show why transfer patterns are robust. It is still possible that the optimal paths remain rather unchanged, for example when the delay is so small that a trip reaches the same connections as without delay. To examine the robustness of transfer patterns in more detail, we conduct another series of experiments and directly delay the optimal routes.

In a first setup, random queries are issued on the New York City data set. For every resulting optimal route one of its conducting trips is delayed with 30 minutes, such that the delay definitely affects the optimal routes. Then the query is repeated on the delayed data and the response is classified as in Section 5.1. Repeating this experiment for more than 32 000 queries showed that only 1.34% of the queries become suboptimal if we influence the connections of the optimal routes in this way.

In order to get a better understanding of the robustness, we extend this experiment to multiple rounds: A random query is drawn as before. In each round, the response of transfer pattern routing is compared to the reference response. Then, one trip of each optimal route is delayed by 30 minutes. This is repeated until the response becomes suboptimal, at most for nine times. Figure 3 summarizes the number of executed iterations until the response became suboptimal. The majority of queries was still optimal after nine successive delays. Also in this setting the transfer patterns computed on the original network proved to be very robust.

## 5.3 Dependencies of the Robustness

Consider an arbitrary query $X@t \to Y$. Let $r^0 = \{p_1^0, p_2^0, \dots\}$ denote the set of paths found by transfer pattern routing in the baseline NULL. In the delayed scenario, the response is $r = \{p_1, p_2, \dots\}$ and $r^* = \{p_1^*, p_2^*, \dots\}$ is the (guaranteed optimal) response of the reference algorithm. The transfer patterns are robust if $r = r^*$ in terms of path costs. An *alternative* *path* is a path $p_i \in r$ with costs equal to the costs of a reference path $p_i^* \in r^*$ and $p_i \neq p_i^0$ in terms of the transfer stations. How come such alternative paths are contained in the query graph? This is because in the precomputation, during the course of time paths of different transfer pattern are optimal for a station pair $A, B$. Besides, the query graph is a digraph with one node (-pair) for each station and can therefore contain further alternatives. For illustration, consider the digraph build from the patterns $A \to B \to C \to D$ and $A \to C \to B \to D$. In addition to the patterns it is created from, it also contains the paths $ABD$ and $ACD$. This effect increases with growing number of patterns between a station pair, and thus also by building the query graph from patterns to and from important stations.

The number of alternatives depends also on the number of neighboring stations of $X$ and $Y$, as the query graph is built from all patterns between these stations. The observations for both data sets in Section 5.1 differ. For Toronto, there are more suboptimal responses and they deviate more from the optimum. Here, the average number of neighbor stations $|S|^{-1} \cdot \sum_{s \in S} |\mathcal{N}(s)|$ is 50, whereas for New York City it is 92. To investigate this further, we select the ten percent of stations with the most and with the fewest neighbor stations of New York City and answer queries as in Section 5.1, but with locations $X$, $Y$ drawn from one of the groups. The results clearly express a difference. Queries in the group with 152 to 306 neighbor stations are less often answered suboptimally than in the group with 1 to 36 neighbors (for Mix Chaos: 0,61% vs. 3.11%). As the maximum walking distance influences the size of the neighborhood, increasing this parameter will probably further improve the robustness.

In summary, transfer patterns allow for alternative routes. When the optimal path is iteratively delayed, at some point the optimum switches to a path with another pattern. Figure 4 shows some examples how the transfer pattern of the optimal path evolves, if the trips along the optimal path are subsequently delayed.

## 5.4 Improving the Robustness

The routing algorithm yields suboptimal responses whenever the optimal path is not contained in the query graph or the overlaid transfer patterns respectively. We studied reasons why the optimal paths in case of delay cannot be found in the overlaid patterns. When there is no delay, these paths are typically just slightly dominated by other paths.

The arrival-chain algorithm described in Section 3.2 selects a dominant subset among the paths between two stations. In a first approach to improve the robustness of the transfer patterns, we relaxed the domination relation for travel times in the arrival-chain algorithm: A cost-tuple $a$ dominates another tuple $b$, if its travel time increased by 5%/10%/20% or at least 2/2/5 minutes is less than that of $b$. Other than expected, the resulting patterns are only slightly more robust, whereas even in the first setting the number of patterns has doubled. This would slow down the search time. Provided that the suboptimal responses are only a few, this minor improvement does not seem worth the additional effort.

To motivate our next approach, consider an optimal path with the transfer pattern $U \to V \to W \to X$ and imagine the trip serving $V \to W$ is delayed. We observed that some of the not-found optimal paths take redirections over some station $R$, but otherwise use parts of the original pattern, for example *UVRX* or *URWX*. As described in Section 5.3, overlaying transfer patterns generates a graph which contains additional paths. We tried to exploit this by enhancing the query graph with additional patterns. In the example, we would add transfer patterns for the station pairs *U, W* and *V,X* hoping that this adds subpaths *VRX* or *URW* to the query graph. While this works in theory, in practice the trigger of this extension remains unclear. Extending the query graph for every delayed arc is impractical, as this

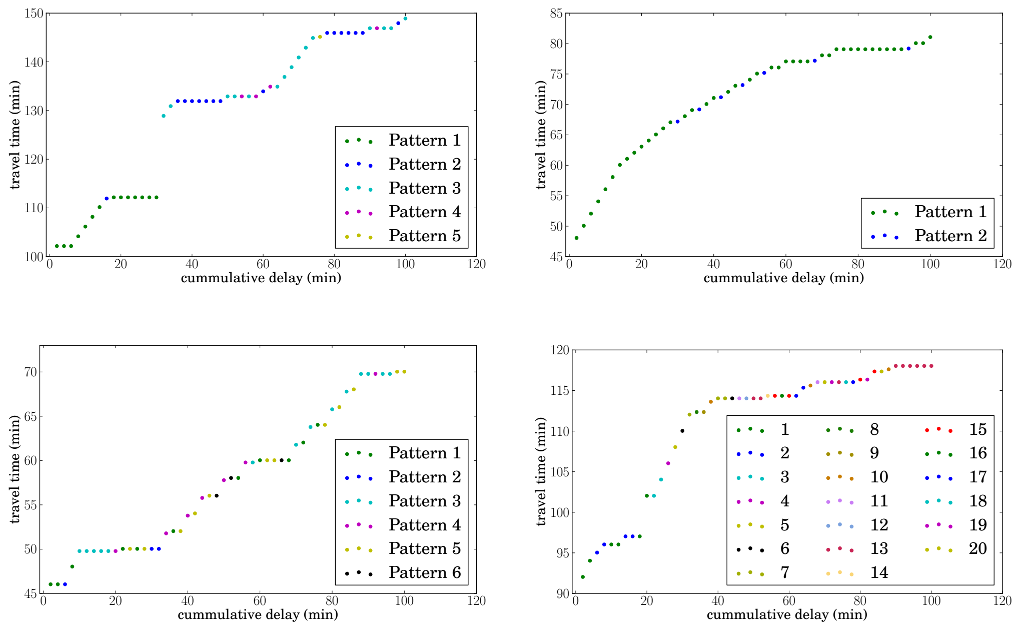


**Figure 4** Evolution of the pattern of an optimal path. From the response to a fixed query, the path with the highest number of transfers is selected and one of its connections is iteratively delayed with two minutes. The plots show how the travel time increases and the pattern is occasionally changed for four exemplary queries.

would blow up its size and the construction time. Triggering the extension only for arcs with delay above a fixed threshold does not reflect the fact that occurrence and severity of suboptimal paths are only weakly related to the amount of delay. Another idea is to repeat the search on the query graph whenever the first search yields a path over a delayed connection. Alternative routes for this connection would be added to the graph. However, the suboptimal paths often do not go via delayed connections, so this is unreliable, too.

## 6 Conclusion & Future Work

We described how delays can be handled by transfer pattern routing without repeating its expensive precomputation. We showed how the data structure for efficient direct connection queries can be updated fast, allowing to adapt to updates in real-time. It transpired that our approach sustains the high quality of results even under extreme delay scenarios. Just a few paths are suboptimal, most of which do not deviate too much from the optimum. For example, when delaying every trip on the New York City data set with 5–50 minutes in average and answering 50 000 queries, only 450 of the resulting paths are not optimal. More than 75% of these are less than 10% and less than ten minutes off the optimum. Furthermore, we provided insight why the transfer patterns contain alternative routes and we analyzed on which factors the robustness depends.

The inherent disadvantage of the scenarios is that they model delay independently. On the one hand, in realistic public transportation delay is often systemic. For example, a traffic jam will delay a series of trips. On the other hand, there are mechanisms to compensate delay: a bus can drive faster to catch up with its schedule. Another example are traffic agencies in the EU, which are bound by law to reimburse passengers for excessive delay. Because of this, the agencies employ decision algorithms which can make connections wait for delayed trains. As delay occurs not independently as assumed in this paper, the acquisition of realistic delay data and repetition of the experiments on top of that is a topic for future research.

Although the quality of responses are almost always optimal, there are some critically suboptimal paths. Future work should focus on eliminating these or making them less severe, for example by adding alternative transfer patterns for frequently delayed trips. We proposed three improvement approaches and discussed why they fail. If a detection mechanism for such bad responses can be found, a fall-back algorithm [3, 4, 5] could be used to find optimal responses on the updated transportation network. In order to be practicable, such a detection must not increase the running-time for the majority of queries, which are already answered optimally. This seems to be a hard problem.

## References

1 Hannah Bast, Erik Carlsson, Arno Eigenwillig, Robert Geisberger, Chris Harrelson, Veselin Raychev, and Fabien Viger. Fast Routing in Very Large Public Transportation Networks Using Transfer Patterns. In *ESA (1)*, volume 6346 of *LNCS*, pages 290–301. Springer, 2010.

2 Annabell Berger, Andreas Gebhardt, Matthias Müller-Hannemann, and Martin Ostrowski. Stochastic Delay Prediction in Large Train Networks. In *ATMOS*, pages 100–111, 2011.

3 Annabell Berger, Martin Grimmer, and Matthias Müller-Hannemann. Fully Dynamic Speed-Up Techniques for Multi-criteria Shortest Path Searches in Time-Dependent Networks. In *SEA*, volume 6049 of *LNCS*, pages 35–46. Springer, 2010.

4 Daniel Delling, Thomas Pajor, and Renato F. Werneck. Round-Based Public Transit Routing. In *ALENEX*, pages 130–140. SIAM / Omnipress, 2012.

5 Julian Dibbelt, Thomas Pajor, Ben Strasser, and Dorothea Wagner. Intriguingly Simple and Fast Transit Routing. In *SEA*, volume 7933 of *LNCS*, pages 43–54. Springer, 2013.

6 Robert Geisberger. *Advanced Route Planning in Transportation Networks.* PhD thesis, Karlsruhe Institute of Technology, 2011.

7 Marc Goerigk, Martin Knoth, Matthias Müller-Hannemann, Marie Schmidt, and Anita Schöbel. The Price of Robustness in Timetable Information. In *ATMOS*, pages 76–87, 2011.

8 Andrew J. Higgins and Erhan Kozan. Modeling Train Delays in Urban Networks. *Transportation Science*, 32(4):346–357, 1998.

9 Mohammad H. Keyhani, Mathias Schnee, Karsten Weihe, and Hans-Peter Zorn. Reliability and Delay Distributions of Train Connections. In *ATMOS*, pages 35–46, 2012.

10 Matthias Müller-Hannemann and Mathias Schnee. Efficient Timetable Information in the Presence of Delays. In *Robust and Online Large-Scale Optimization*, volume 5868 of *LNCS*, pages 249–272. Springer, 2009.

11 Evangelia Pyrga, Frank Schulz, Dorothea Wagner, and Christos D. Zaroliagis. Efficient Models for Timetable Information in Public Transportation Systems. *ACM Journal of Experimental Algorithmics*, 12, 2007.

12 Jonas Sternisko. On Compact Representation and Robustness of Transfer Patterns in Public Transportation Routing. Master's thesis, Universität Freiburg, April 2013.

13 Jianxin Yuan. *Stochastic Modelling of Train Delays and Delay Propagation in Stations.* PhD thesis, Technische Universiteit Delft, The Netherlands, 2006.

# Solving a Freight Railcar Flow Problem Arising in Russia

Ruslan Sadykov*[1], Alexander A. Lazarev[2,3], Vitaliy Shiryaev[4], and Alexey Stratonnikov[4]

1 INRIA Bordeaux – Sud-Ouest,
351, cours de la Liberation, 33405 Talence, France
Ruslan.Sadykov@inria.fr

2 Institute of Control Sciences,
65 Profsoyuznaya street, 117997 Moscow, Russia
lazarev@ipu.ru

3 National Research University Higher School of Economics,
20 Myasnitskaya street, 101000 Moscow, Russia

4 JSC Freight One
Staraya Basmannaya st., 12 bld. 1, 105064 Moscow, Russia
{ShiryaevVV,StratonnikovAA}@pgkweb.ru

## Abstract

We consider a variant of the freight railcar flow problem. In this problem, we need 1) to choose a set of transportation demands between stations in a railroad network, and 2) to fulfill these demands by appropriately routing the set of available railcars, while maximizing the total profit. We formulate this problem as a multi-commodity flow problem in a large space-time graph. Three approaches are proposed to solve the Linear Programming relaxation of this formulation: direct solution by an LP solver, a column generation approach based on the path reformulation, and a "column generation for extended formulations" approach. In the latter, the multi-commodity flow formulation is solved iteratively by dynamic generation of arc flow variables. Three approaches have been tested on a set of real-life instances provided by one of the largest freight rail transportation companies in Russia. Instances with up to 10 millions of arc flow variables were solved within minutes of computational time.

**1998 ACM Subject Classification** G.1.6 Optimization

**Keywords and phrases** Freight routing, multi-commodity flow, column generation

**Digital Object Identifier** 10.4230/OASIcs.ATMOS.2013.55

## 1 Introduction

In Russia, the activity of forming and scheduling freight trains is separated by a regulation from the activity of managing the fleet of freight railcars. A state company is in charge of the first activity. Freight railcars are owned by several independent companies. Every such company is quite limited in transportation decisions due to the separation of activities. A company which owns a fleet of railcars can only accept or refuse a transportation demand. Then it must assign railcars to accepted demands. In some cases, the company has a possibility to slightly modify the execution date of a demand, which gives more flexibility to the decision process but makes it more complicated.

* Corresponding author

Thus, an operational plan of such a company is determined by 1) a set of accepted transportation demands, 2) for each demand, its execution date and the set of cars assigned to it, and 3) empty cars movements to supply each demand. As the company is commercial, a reasonable criterion for the quality of an operational plan is the profit generated by it. The profit is determined by the difference between the price collected for fulfilling transportation demands and the costs paid to the state company for exploiting the railroad network.

In this paper, we study the problem of finding a most profitable operational plan for a company which owns and manages a fleet of railcars. This problem was formulated by the mathematical modeling department of one of the largest such companies in Russia.

For this problem, we are given a railroad network, a set of transportation demands, an initial location of cars of different types. The network data consists of a set of stations, travel times and costs between them. As it was mentioned above, the company does not schedule trains. Thus, actual transportation of loaded and empty railcars is performed by the state company, who charges predetermined costs per trip. Estimated travel times are also determined and applied by the state company. Note that the transfer cost for an empty car depends on the type of product type loaded previously to this car. This way, the state increases attractiveness of the transportation of "socially important" products (for example, coil). This custom is being vanished now, but it is still practiced for some car types.

The objective is, for a given period of time, to choose a set of demands to be met, totally or partially, and, for each car, find a route which includes both loaded and empty transfers.

To solve the problem, we start from an integer multi-commodity flow model, which has been proposed by Stratonnikov and Shiryaev [10]. The time horizon in this model is discretized in periods of one day. This discretization choice is reasonable for Russia, as distances are measured in thousands of kilometers, and the average speed of freight trains is relatively low: about 300 kilometers per day (it tends to decrease further with a saturation of the network).

This model has a very large size, and even solving its Linear Programming (LP) relaxation using modern commercial solvers can take hours of computation time for real-life instances. However, a solution of this LP relaxation allows one to obtain a very tight dual bound for the objective function value. This fact has been also noticed for similar models considered in [4, 7]. Therefore, we concentrate on solving the LP relaxation of this formulation, leaving the problem of obtaining an integer solution out of the scope of the paper. In practice, rounding a fractional solution in a straightforward way allows one to obtain an integer solution with very small gap.

To solve the LP relaxation faster, we devise two variants of the column generation procedure, where columns represent railcar routes or flows of the railcars of the same type. The first variant we tried is the classic Dantzig-Wolfe approach. In the second variant, called "column generation for extended formulations" in [9], columns are disaggregated into individual arc flow variables when added to the master. Thus, the master problem is equivalent to the original multi-commodity flow model, but its variables are generated dynamically. On almost all real-life instances provided by the company, either the first or the second variant of the column generation approach significantly outperformed the solution by a solver of the LP relaxation of the original multi-commodity model, preprocessed by a problem-specific procedure.

To our knowledge, the closest model considered in the literature is the freight car flow problem faced by a Brazilian logistics operator and described by Fukasawa et al. [4]. In this paper, authors proposed a similar integer multi-commodity flow model and solved it using a simple preprocessing and an Mixed Integer Programming (MIP) solver. The main difference with our model is the availability of a fixed train schedule. In their model cars must be assigned to trains to be transported. In our model, we cannot rely on the train schedule information, as it is very approximative and rarely respected in practice. Instead, we use the normative travel times given by the state company which is in charge of forming and scheduling trains. Additionally, Fukasawa et al. considered loading and unloading times, which we neglect here because they are much smaller than the length of time periods.

Another similar car flow model has been considered by Holmberg et al. [5]. In this model, one searches only for a flow of empty cars, the flow of loaded cars being fixed. Thus, a heuristic iterative procedure is applied to optimize the total flow of cars.

A paper which is related to our research in terms of the solution approach applied is due to Löbel [7], who considered a vehicle scheduling problem arising in public mass transit. This problem is modeled by a multi-commodity flow model formulation, the LP relaxation of which is solved by dynamically generating arc variables, as we do. With this approach, Löbel was able to solve LP relaxations with millions of variables as we do for the freight car flow problem.

## 2 Problem description

We give a detailed description of the variant of the freight car flow model considered here. The problem is to find a feasible flow of railcars (i.e. a feasible route for each car) that maximizes the profit by meeting a subset of the transportation demands.

The railroad network consists of a set of stations. Travel times and costs are known for each "origin-destination" pair of stations. Times are measured in days and rounded up. The cost for an empty car transfer depend on the type of the latest product this car has transported, as explained in the introduction.

Number of cars, their initial locations and availability dates are known. Cars are divided into types. The type of a car determines types of products which can be loaded on this car. The route of a car consists of a sequence of alternating loaded and empty movements between stations. Cars can wait at stations before and after fulfilling transportation demands. In this case, a charge is applied. Daily rate of this charge depends on the demand before (or after) the waiting period.

Each transportation demand is defined by a (maximum) number of cars compatible with the product that should be taken from an origin station to a destination station. Some demands can be fulfilled partially. In this case, the client communicates the minimum number of cars which should be delivered. Thus, the total number of transported cars for every accepted demand should be between the minimum and maximum number.

The client specifies the availability date of the product and the delivery due date which cannot be exceeded. The demand transportation time is known. This allows us to determine the latest date at which the transportation must start. The profit we gain for meeting the demand depends on the date the transportation of a loaded car starts. In practice, the contract is concluded for transportation of each car separately. Thus the profit we gain for delivering cars with the product of a same demand at a certain date depends linearly on the number of cars. Note that the profit function already takes into account the charges paid for using the railroad network.

We now specify notations for the data of the problem. Following sets are given.

- $I$ — set of stations.
- $C$ — set of car types.
- $K$ — set of product types

- $Q$ — set of demands
- $S$ — set of “sources” which specify initial state of cars.
- $T$ — set of periods (planning horizon).

For each station, $i \in I$ we know sets $W_i^1$ and $W_i^2$ of standing daily rates for cars waiting to be loaded and waiting after unloading.

For each demand $q \in Q$ we know:

- $i_q \in I$ — origin station
- $j_q \in I$ — destination station
- $k_q \in K$ — type of product to be transported
- $C_q \subseteq C$ — set of car types, which can be used for this demand
- $n_q^{\max}$ — number of cars needed to fullfil the demand
- $n_q^{\min}$ — minimum number of car needed to partially fullfil the demand
- $r_q \in T$ — demand availability, i.e. the period starting from which the transportation of the product can start
- $\Delta_q$ — maximum delay for starting the transportation
- $\rho_{qt}$ — profit from delivery of one car with the product, transportation of which started at period $t$, $t \in [r_q, r_q + \Delta_q]$
- $d_q \in \mathbb{Z}_+$ — transportation time of the demand
- $w_q^1 \in W^1_{i^1_q}$ — daily standing rate charged for one car waiting before loading the product at origin station
- $w_q^2 \in W^2_{i^2_q}$ — daily standing rate charged for one car waiting after unloading the product at destination station

For each car type $c \in C$, we can obtain set $Q_c$ of demands, which a car of type $c$ can fulfill.

For each source $s \in S$, we are given:

- $\vec{i}_s \in I$ — station where cars are located
- $\vec{c}_s \in C$ — type of cars
- $\vec{r}_s \in T$ — period, starting from which cars can be used
- $\vec{w}_s \in W^2_{\vec{i}_s}$ — daily standing rate charged for cars
- $\vec{k}_s \in K$ — type of the latest delivered product
- $\vec{n}_s \in \mathbb{N}$ — number of cars in the source

For each car type $c \in C$, we can obtain set of sources $S_c = \{s \in S : \ \vec{c}_s = c\}$.

Additionally, functions $M(c, i, j, k)$ and $D(c, i, j)$ are given which specify cost and duration of transportation of one empty car of type $c \in C$ from station $i \in I$ to station $j \in I$ under condition, that the type of the latest delivered product is $k \in K$ (for the cost).

## 3 Mathematical model

We represent movements of cars of each type $c \in C$ by commodity $c$. For each commodity $c \in C$, we introduce a directed graph $G_c = (V_c, A_c)$. Set $V_c$ of vertices is divided into two subsets $V_c^1$ and $V_c^2$ which represent respectively states in which cars stand at a station before being loaded and after being unloaded. A vertex $v_{cit}^{1w} \in V_c^1$ represents stay of cars of type $c$ waiting to be loaded at station $i \in I$ at daily rate $w \in W_i^1$ at period $t \in T$. Flow balance $b(v_{cit}^{1w})$ of this vertex is zero. A vertex $v_{cit}^{2wk} \in V_c^2$ represents stay of cars of type $c$ after being unloaded at station $i \in I$ at daily rate $w \in W_i^2$ at period $t \in T$. Here $k \in K$ is the type of unloaded product. Flow balance $b(v_{cit}^{2wk})$ of this vertex is determined as follows:

$$b(v_{cit}^{2wk}) = \begin{cases} \vec{n}_s, & \exists s \in S_c : \vec{i}_s = i, \vec{r}_s = t, \vec{w}_s = w, \vec{k}_s = k, \\ 0, & \text{otherwise.} \end{cases}$$

Additionally, there is a single terminal vertex with flow balance equal to $-\sum_{s\in S_c}\vec{n}_s$.

There are three types of arcs in $A_c$: waiting, empty transfer, and loaded transfer arcs.

- A waiting arc $a^{\alpha wk}_{cit}$ represents waiting of cars of type $c$ from period $t \in T$ to $t+1$ at station $i \in I$ at daily rate $w \in W^\alpha_i$ before being loaded ($\alpha = 1$) or after being unloaded ($\alpha = 2$). $k \in K$ is the type of unloaded product in case $\alpha = 2$. This arc goes from vertex $v^{\alpha wk}_{cit}$ to vertex $v^{\alpha wk}_{c,i,t+1}$, or to the terminal vertex if $t+1 \notin T$. Cost of this arc is $w$.
- An empty transfer arc $a^{w'w''k}_{cijt}$ represents a transfer of empty cars of type $c$ waiting at station $i \in I$ at daily rate $w' \in W^2_i$ to station $j \in I$ where they will wait at daily rate $w'' \in W^1_j$, such that the type of latest unloaded product is $k \in K$, and transfer starts at period $t \in T$. This arc goes from vertex $v^{2w'k}_{cit}$ to vertex $v^{1w''}_{cjt'}$, or to the terminal vertex if $t' \notin T$, where $t' = t + D(c,i,j)$. Cost of this arc is $M(c,i,j,k)$.
- A loaded transfer arc $a_{cqt}$ represents transportation of the product of demand $q \in Q$ by cars of type $c$ starting at period $t \in T \cap [r_q, r_q + \Delta_q]$. This arc goes from vertex $v^{1w^1_q}_{ci_qt}$ to vertex $v^{2w^2_qk_q}_{c,j_q,t+d_q}$, or to the terminal vertex if $\{t+d_q\} \notin T$. The cost of this arc is $-\rho_{qt}$.

A small example of graph $G_c$ is depicted in Figure 1. In this example, there is only one "before" vertex and one "after" vertex for each time period and each station. In real-life examples, there are several rows of "before" and "after" vertices for each station.

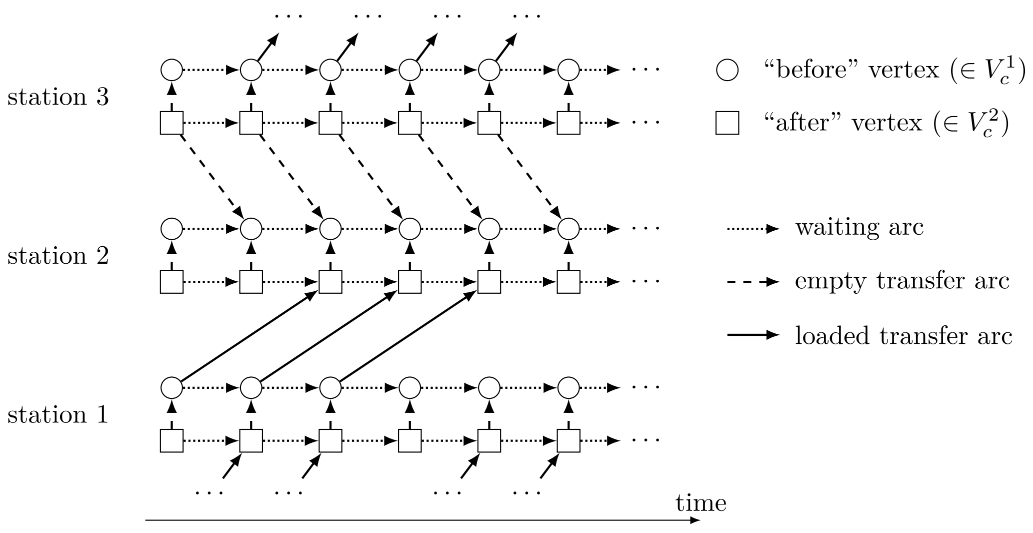


**Figure 1** An example of graph $G_c$

We denote as $A_{cq}$ the set of all loaded transfer acs related to demand $q \in Q_c$: $A_{cq} = \{a_{cq't} \in A_c : q' = q\}$. Also we denote as $\delta^+(v)$ and $\delta^-(v)$ the sets of incoming and outgoing arcs for vertex $v_c$.

From now on, graph $G_c$ is assumed to be trivially preprocessed: we remove vertices with degree two (replacing appropriately incident arcs), and remove every vertex (together with incident arcs) such that there is no path from any source to it or there is no path from it to the terminal vertex.

For each commodity $c \in C$ and for each arc $a \in A_c$, we define an integer variable $x_a$ which represents the flow size of commodity $c$ along arc $a$. Cost of arc $a$ is denoted as $g(a)$. Additionally, for each demand $q \in Q$, we define a binary variable $y_q$ which indicates whether demand $q$ is accepted or not.

Now we are able to present a multi-commodity flow formulation ($MCF$) for the problem.

$$\min \sum_{c\in C} \sum_{a\in A_c} g(a)x_a \tag{1}$$

$$\sum_{c\in C_q} \sum_{a\in A_{cq}} x_a \le n_q^{\max} y_q \quad \forall q \in Q \tag{2}$$

$$\sum_{c\in C_q} \sum_{a\in A_{cq}} x_a \ge n_q^{\min} y_q \quad \forall q \in Q \tag{3}$$

$$\sum_{a\in\delta^-(v)} x_a - \sum_{a\in\delta^+(v)} x_a = b(v) \quad \forall c \in C, v \in V_c \tag{4}$$

$$x_a \in \mathbb{Z}_+ \quad \forall c \in C, a \in V_c \tag{5}$$

$$y_q \in \{0,1\} \quad \forall q \in Q \tag{6}$$

Constraints (2) and (3) specify that the number of cars assigned to accepted demand $q$ should be between $n_q^{\min}$ and $n_q^{\max}$. Constraints (4) are flow conservation constraints for each commodity. As formulation $(MCF)$ generalizes the standard multi-commodity flow problem (where variables $y$ are fixed to one), our problem is NP-hard in the strong sense.

The formulation $(MCF)$ was tested in [10]. The main difficulty was to solve the LP relaxation of the problem, which we denote as $(MCF)_{LP}$. Even after non-trivial problem-specific preprocessing, solution time of $(MCF)_{LP}$ for typical real-life instances by a modern LP solver on a modern computer is more than one hour. Therefore, our research is concentrated on accelerating the resolution of $(MCF)_{LP}$.

## 4 A column generation approach

A classic approach to solve multi-commodity flow formulations is to apply the column generation procedure. Instead of working with arc variables, one uses variables of one the following types.

- A "path variable" determines the flow size of a commodity along a path from one of the source nodes to a sink node of this commodity.
- A "tree variable" specifies whether the flow of a certain size from a single source of a commodity goes along a fixed (directed) tree with fixed flow sizes along its arcs. Leaves of this tree are sink nodes of this commodity.
- A "flow enumeration variable" specifies whether the flow of a single commodity is equal to a fixed flow.

We now reformulate $(MCF)$ using path variables, and then using flow enumeration variables. The reformulation which uses the tree variables was not tried, as there is only one sink per commodity.

### 4.1 Path reformulation

For each commodity $c \in C$ and each source $s \in S_c$, we denote as $P_s$ the set of paths going from the corresponding source vertex in $V_c$ to the terminal vertex of graph $G_c$. Each such path represents a route for cars originating at source $s$. For a path $p \in P_s$, we introduce a variable $\lambda_p$ which determines the flow size along path $p$ (or number of cars taking this route). Let $A_p^{path}$ be the set of arcs taken by a path $p \in P_s$ and $g_p^{path}$ be the cost of the path: $g_p = \sum_{a\in A_p^{path}} g(a)$. Let also $Q_p^{path}$ be the set of demands "covered" by path $p$. The path reformulation $(PTH)$ of $(MCF)$ is the following.

$$\min \sum_{c\in C}\sum_{s\in S_c}\sum_{p\in P_s} g_p^{path}\lambda_p \tag{7}$$

$$\sum_{c\in C_q}\sum_{s\in S_c}\sum_{p\in P_s\colon q\in Q_p^{path}} \lambda_a \le n_q^{\max} y_q \quad \forall q\in Q \tag{8}$$

$$\sum_{c\in C_q}\sum_{s\in S_c}\sum_{p\in P_s\colon q\in Q_p^{path}} \lambda_a \ge n_q^{\min} y_q \quad \forall q\in Q \tag{9}$$

$$\sum_{p\in P_s} \lambda_p = \vec{n}_s \quad \forall c\in C, s\in S_c \tag{10}$$

$$\lambda_p \in \mathbb{Z}_+ \quad \forall c\in C, s\in S_c, p\in P_s$$

$$y_q \in \{0,1\} \quad \forall q\in Q$$

Constraints (8) and (9) are rewritten constraints (2) and (3). Constraints (10) guarantee that a route is assigned to every car in each source.

In order to solve the LP relaxation $(PTH)_{LP}$ of formulation $(PTH)$, we apply the column generation procedure. On each iteration of it, the formulation $(PTH)_{LP}$ with a restricted number of variables $\lambda$ (which we will call the restricted master) is solved, and optimal primal and dual solutions are obtained. Let $\pi^{\max}$, $\pi^{\min}$, and $\mu$ be the vectors of optimal dual solution values corresponding to constraints (8), (9), and (10). Then the pricing problem is solved which determines whether there exists a variable with a negative reduced cost absent from the restricted master. The reduced cost $\bar{g}_p^{path}$ of a variable $\lambda_p$, $p\in P_s$, $s\in S_c$, $c\in C$ is computed as

$$\bar{g}_p^{path} = \sum_{a\in A_p^{path}} g(a) + \sum_{q\in Q_p^{path}} (\pi_q^{\max} - \pi_q^{\min}) - \mu_s. \tag{11}$$

The problem of finding a variable $\lambda$ with the minimum reduced cost can be solved by a sequence of shortest path problems between each source $s\in S_c$ and the terminal vertex for every commodity $c\in C$. To accelerate the solution of the pricing problem, instead of searching the shortest path separately for each source, in each graph $G_c$, we can find a minimum cost in-tree to the terminal vertex from every source in $S_c$. As directed graphs $G_c$ are acyclic (each arc except those from $V_c^2$ to $V_c^1$ induces a time increase), the complexity of this procedure is linear in the number of arcs for each graph $G_c$.

This procedure is quite fast, but its disadvantage consists in significant demand "overcovering". This means that many generated paths contain arcs corresponding to same demands, i.e. much more cars are assigned to these demands than needed. This has a bad impact on the convergence of column generation.

Therefore, we developed an iterative procedure which heuristically constructs a solution to the original problem with demand profits modified by the current dual solution values. Then all paths which constitute this solution are added to the master. On each iteration, we search for a shortest path tree and then remove covered demands and cars assigned to them for the next iteration. The heuristic stops when either all demands are covered, or all cars are assigned, or maximum number of iterations is reached. The latter is a parameter which we denote as `nbPricIter`. This procedure for commodity $c\in C$ is formally presented in Algorithm 1. This procedure can be viewed as a heuristic for generating additional paths to be added to the master for the convergence acceleration.

**Algorithm 1:** Path generation iterative pricing procedure for graph $G_c$

```
foreach demand $q \in Q_c$ do $uncovCars_q \leftarrow n_q^{\max}$;
foreach source $s \in S_c$ do $remCars_s \leftarrow \vec{n}_s$;
$iter \leftarrow 0$;
repeat
    Find an in-tree to the terminal from sources $s \in S_c$, $remCars_s > 0$;
    Sort paths $p$ in this tree by non-decreasing of their reduced cost $\bar{g}_p^{path}$;
    foreach path $p$ in this order do
        $minCars \leftarrow \min\{uncovCars_q \mid q \in Q_p^{path}\}$;
        if $\bar{g}_p < 0$ and $minCars > 0$ then
            Add variable $\lambda_p$ to the restricted master;
            $s \leftarrow$ the source of $p$;
            $remCars_s \leftarrow remCars_s - \min\{remCars_s, minCars\}$;
            foreach $q \in Q_p^{path}$ do
                $uncovCars_q \leftarrow uncovCars_q - \min\{remCars_s, minCars\}$;
    $iter \leftarrow iter + 1$;
until $uncovCars_q = 0$, $\forall q \in Q_c$, or $remCars_s = 0$, $\forall s \in S_c$, or $iter =$nbPricIter;
```

## 4.2 Flow enumeration reformulation

Each car type defines a commodity $c \in C$. We define by $F_c$ the set of all fixed solutions (fixed flows) for commodity $c$. For a flow $f \in F_c$, we introduce a binary variable $\omega_f$ which specifies whether cars of type $c$ are routed according to flow $f$ or not. Let $f_a$ be the size of flow $f$ along arc $a \in A_c$ and $g_f^{flow}$ be the cost of the flow: $g_f^{flow} = \sum_{a \in A_c} f_a \cdot g(a)$. The commodity reformulation $(FEN)$ of $(MCF)$ is the following

$$\min \sum_{c \in C} \sum_{f \in F_s} g_f^{flow} \omega_f \tag{12}$$

$$\sum_{c \in C_q} \sum_{f \in F_c} \sum_{a \in A_{cq}} f_a \omega_f \leq n_q^{\max} y_q \quad \forall q \in Q \tag{13}$$

$$\sum_{c \in C_q} \sum_{f \in F_c} \sum_{a \in A_{cq}} f_a \omega_f \geq n_q^{\min} y_q \quad \forall q \in Q \tag{14}$$

$$\sum_{f \in F_c} \omega_f = 1 \quad \forall c \in C \tag{15}$$

$$\omega_f \in \{0,1\} \quad \forall c \in C, f \in F_c$$

$$y_q \in \{0,1\} \quad \forall q \in Q$$

Constraints (13) and (14) are rewritten constraints (2) and (3). Constraints (15) guarantee that exactly one flow is assigned to commodity $c \in C$.

LP relaxation $(FEN)_{LP}$ of formulation $(FEN)$ can also be solved by the column generation procedure. The pricing problem here decomposes into the minimum cost flow problems for each commodity $c \in C$.

Our computational results showed that, solving $(FEN)_{LP}$ by column generation is not practical due to convergence problems. However, in the next section, we present a modification of this approach, which is computationally much more efficient.

## 5 A "column generation for extended formulations" approach

We adapt here the hybrid approach, reviewed under the name "column generation for extended formulations" (CGEF) in [9]. The idea is to solve formulation $(MCF)_{LP}$ iteratively by generating arc flow variables dynamically. On each iteration, we generate columns (single commodity flows) for formulation $(FEN)_{LP}$, and translate them into arc flow variables which are added to formulation $(MCF)_{LP}$.

In the CGEF approach, on each iteration, we first solve the formulation $(MCF)_{LP}$ with a restricted number of variables $x$. We will also call this formulation the restricted master. Then, we verify whether there are variables $x$ with a negative reduced cost absent from the restricted master. However, we do not do it by enumeration, but by using the same pricing problem as in classic column generation for solving formulation $(FEN)_{LP}$. If a pricing problem solution with a negative reduced cost is found, we add to the restricted master variables $x$ which are positive in this solution (some of them can be already in the restricted master).

As a consequence of the theorem proved in [9], we know that, if an arc flow variable $x$ is absent from the restricted master and has a negative reduced cost in the current solution of the restricted master, there exists a pricing problem solution with a negative reduced cost where this variable is positive. Therefore, if there are no pricing problem solutions with a negative reduced cost, the current solution of the restricted master is optimal for $(MCF)_{LP}$.

When solving formulation $(MCF)_{LP}$ by the CGEF approach, the pricing problem is decomposed to a sequence of min-reduced-cost flow problems for each commodity $c \in C$, as in the column generation approach for solving the commodity reformulation $(FEN)_{LP}$. Let $\pi_q^{\max}$ and $\pi_q^{\min}$ be the vectors of optimal dual solution values corresponding to constraints (2), (3). Then, the reduced cost $\bar{g}_f^{flow}$ of a flow $f \in F_c$ is computed as

$$\bar{g}_f^{flow} = \sum_{a \in A_c} f_a \cdot g(a) + \sum_{q \in Q_c} \sum_{a \in A_{cq}} f_a \cdot (\pi_q^{\max} - \pi_q^{\min}). \tag{16}$$

Note that dual values corresponding to flow conservation constraints (4) are not taken into account when solving the pricing problem, as explained in [9]: this follows from the fact that these constraints are satisfied by the pricing problem solution.

On each iteration, for each commodity $c \in C$, the pricing problem generates a flow $f \in F_c$ with the minimum reduced cost. If this cost is negative, variables $x$ corresponding to arcs on which flow $f$ is positive, are added to the restricted master. Otherwise, the current solution of the restricted master is optimal for $(MCF)_{LP}$, and we stop.

## 6 Numerical results

The test instances were provided to us by the mathematical modeling departement of *JSC Freight One*, which is one of the largest freight rail transportation companies in Russia.

We have numerically tested the following three approaches for solving formulation $(MCF)_{LP}$ on these real-life instances.

1. Direct solution of $(MCF)_{LP}$ by the *Clp* LP solver [1]. Before applying the LP solver the formulation is preprocessed by a non-trivial problem specific procedure. This procedure is not public and it was not available to us. Moreover, the open-source solver *Clp* was specifically modified to better tackle formulation $(MCF)_{LP}$. Thus, this approach was applied inside the company. We tried to solve $(MCF)_{LP}$ with only trivial preprocessing by the default version of both LP solvers *Clp* and *Cplex* [2], but our solution times on a

comparable computer were significantly larger. Therefore, for the comparison, we use the solution times communicated to us by the company. We denote this approach as Direct.

2. Solution of the path reformulation $(PTH)_{LP}$ by column generation. The pricing problem here is solved by the iterative shortest path tree procedure presented in Algorithm 1. The effect of applying the iterative pricing procedure was significant. After preliminary tests, the parameter `nbPricIter` was set to 5. For better convergence of the column generation procedure, the following improvements are applied.
   - The restricted master is initialized with paths according to which cars stay at their initial locations during all the planning horizon.
   - Stabilization by dual prices smoothing [8] is applied.
   - The restricted master is cleaned up every 10 iterations by deleting all columns with a positive reduced cost.

   The column generation approach was implemented in C++ programming language using the *BaPCod* library [11] and *Cplex* as LP solver. We denote this approach as ColGen.
3. The solution of $(MCF)_{LP}$ by the CGEF approach. The pricing problem here is solved using the minimum cost flow solver *Lemon* [3]. To improve convergence of the algorithm, the master is initialized with the full set of waiting arcs. Note that in distinction to Direct only a trivial procedure was applied to preprocess the formulation. This approach was also implemented in the same manner as the previous one. We denote this approach as ColGenEF.

The approach Direct was run on a computer with a processor Intel Xeon X5677 3.47 GHz, the approaches ColGen and ColGenEF were run on a computer with a processor Intel Xeon X5460 3.16 GHz in a single thread mode.

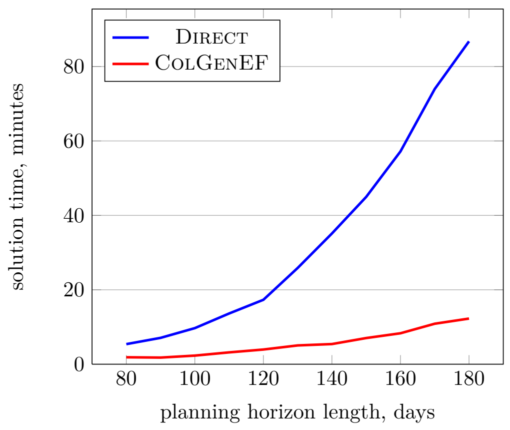
The first test set consists of 3 instances. Characteristics of these instances and results for 3 tested approaches for these instances are presented in Table 1.

The difference in performance of the approaches Direct and ColGenEF on instances `x3` and `x3double` can be explained by the problem specific preprocessing. Although we are not aware of preprocessing details, we know that it is based on similarities between car types. For instance `5k0711q` in which there is only one car type, difference between two approaches is much smaller. Note that this instance has been artificially created from the real-life one by merging car types into one.

The second test set consists of instances with larger planning horizon length. These instances contain 1'025 stations, up to 6'800 demands, 11 car types, 12'651 cars, and 8'232

**Table 1** The first set of instances: characteristics and numerical results.

| Instance name | `x3` | `x3double` | `5k0711q` |
|---|---|---|---|
| Number of stations | 371 | 371 | 1'900 |
| Number of demands | 1'684 | 3'368 | 7'424 |
| Number of car types | 17 | 17 | 1 |
| Number of cars | 1'013 | 1'013 | 15'008 |
| Number of sources | 791 | 791 | 11'215 |
| Time horizon, days | 37 | 74 | 35 |
| Total number of vertices, thousands | 62 | 152 | 22 |
| Total number of arcs, thousands | 794 | 2'846 | 1'843 |
| Solution time for Direct | 20s | 1h34m | 55s |
| Solution time for ColGen | 22s | 7m53s | 8m59s |
| Solution time for ColGenEF | 3m55s | >2h | 43s |



| Horizon | Direct | ColGenEF |
|---|---|---|
| 80 | 5m24s | 1m52s |
| 90 | 7m05s | 1m47s |
| 100 | 9m42s | 2m19s |
| 110 | 13m38s | 3m11s |
| 120 | 17m19s | 3m57s |
| 130 | 25m52s | 5m03s |
| 140 | 35m08s | 5m25s |
| 150 | 44m58s | 7m02s |
| 160 | 57m11s | 8m19s |
| 170 | 1h13m58s | 10m53s |
| 180 | 1h26m46s | 12m16s |

**Figure 2** Solution times for test instances with larger planning horizon length.

sources. The planning horizon length is from 80 to 180 days. The graph $\cup_{c \in C} G_c$ for the largest instance contains about 300 thousands nodes and 10 millions arcs. For these instances, the two best approaches are Direct and ColGenEF. The comparison of their solution times is presented in Figure 2. The approach ColGen is two to three times slower than Direct.

An important observation is that the algorithm ColGenEF generally converges in less than 10 iterations (and always in less than 15 iterations). The restricted master on the final iteration contains only about 3% of the arc flow variables of formulation $(MCF)$.

## 7 Conclusions and perspectives

We have formulated a freight car flow problem variant as a multi-commodity flow problem in a large space-time graph. Three approaches for solving the LP relaxation of this formulation has been tested on a set of real-life instances provided by one of the largest freight rail transportation companies in Russia.

Computational results show that the classic column generation approach is the best for instances with relatively small number of sources (different initial locations of cars). For other instances, approaches based on the multi-commodity formulation produce better results. Problem-specific preprocessing based on similarities between car types is an important ingredient, which allows a modern LP solver to tackle quite efficiently instances with a relatively small time horizon length. The best approach for instances with larger time horizon length is solving the multi-commodity formulation by dynamically generating arc flow variables (the "column generation for extended formulations" approach). Even without applying problem-specific preprocessing, it outperforms the direct resolution approach, and this advantage increases with the increase of the time horizon length. It is likely that the combination of the CGEF approach with problem-specific preprocessing will produce even better results. Such a combination could be used to produce better solutions by shortening the time period length.

The most important research direction for the future is to obtain integer solutions for the problem either by a branch-and-bound (or branch-and-price) methods or by heuristics based on the fractional solution obtained by the approaches proposed here. Simple heuristics can be based on rounding. Experiments conducted inside the company show that this approach already produces good results, as the LP relaxation solutions are almost integer. Column generation based heuristics of the "diving" type [6] are likely to produce better results.

Note that the problem studied in this paper does not incorporate some practical considerations. Some of them can be easily modeled by enlarging the space-time graph used in the current approaches. These considerations are the following.

- Waiting rates for cars are generally not linear but progressive. When a car arrives to a station, the owner should pay an initial rate for every standing day. After a certain period of time this rate increases. This increase can happen several times. In other words, the daily waiting rate in every station is a non-decreasing function of the current stay duration.
- There is a set of special stations where cars can stay for a lower rate (although there is a fix rate for putting cars there). A car can go to one of this stations between fulfilling two demands in order to pay less for waiting. Note that it is advantageous to use these stations only if a sufficiently long planning horizon is considered.
- There is a compatibility function between two consecutive types of loaded products. This means that even if a car type is suitable for a demand, a car of this type may not be able to fulfill it because the type of previously loaded product is incompatible with the product type of the demand. For example, the petrol cannot be loaded to a car after the oil, but the oil can be put after the petrol. Also, there are special stations where cars can be washed for a fee. After washing a car, the type of product previously loaded to it is "nullified".
- When a demand is not selected, a penalty payment may be due.

There exists however a problem extension which cannot be solved by the approaches presented. As mentioned in the introduction, transportation times and costs between each pair of stations are communicated by the state company which is in charge of forming and scheduling trains. In this paper, we considered that they depend only on the origin and destination stations. However, in practice usually they also depend on the size of the group of cars sent together. The larger this group is, the faster and with smaller unitary cost it will be delivered to the destination. It seems that exact solution of real-life instances of this extension of the problem is out of reach of modern optimization tools.

## References

1 Clp – COIN-OR Linear Programming Solver. `https://projects.coin-or.org/Clp`.

2 IBM ILOG CPLEX Optimizer. `http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/`.

3 LEMON Graph Library. `https://lemon.cs.elte.hu/trac/lemon`.

4 Ricardo Fukasawa, Marcus Poggi de Aragão, Oscar Porto, and Eduardo Uchoa. Solving the freight car flow problem to optimality. *Electronic Notes in Theoretical Computer Science*, 66(6):42–52, 2002. ATMOS 2002.

5 Kaj Holmberg, Martin Joborn, and Jan T. Lundgren. Improved empty freight car distribution. *Transportation Science*, 32(2):163–173, 1998.

6 Cédric Joncour, Sophie Michel, Ruslan Sadykov, Dmitry Sverdlov, and François Vanderbeck. Column generation based primal heuristics. *Electronic Notes in Discrete Mathematics*, 36:695–702, 2010.

7 Andreas Löbel. Vehicle scheduling in public transit and lagrangean pricing. *Management Science*, 44(12):1637–1649, 1998.

8 Artur Pessoa, Ruslan Sadykov, Eduardo Uchoa, and Francois Vanderbeck. In-out separation and column generation stabilization by dual price smoothing. In *12th International Symposium on Experimental Algorithms*, volume 7933 of *Lecture Notes in Computer Science*, pages 354–365. 2013.

9 Ruslan Sadykov and François Vanderbeck. Column generation for extended formulations. *EURO Journal on Computational Optimization*, 1(1-2):81–115, 2013.

10 Alexey Stratonnikov and Vitaly Shiryaev. A large-scale linear programming formulation for railcars flow management (in Russian). In *Fifth Russian conference on Optimization Problems and Economic Applications*, Omsk, Russia, July 2012.

11 François Vanderbeck. BaPCod — a generic Branch-And-Price Code. `https://wiki.bordeaux.inria.fr/realopt/pmwiki.php/Project/BaPCod`.

# A Configuration Model for the Line Planning Problem *

**Ralf Borndörfer, Heide Hoppmann, and Marika Karbstein**

**Zuse Institute Berlin**
**Takustr. 7, 14195 Berlin, Germany**
`{borndoerfer,hoppmann,karbstein}@zib.de`

**Abstract**

We propose a novel extended formulation for the line planning problem in public transport. It is based on a new concept of *frequency configurations* that account for all possible options to provide a required transportation capacity on an infrastructure edge. We show that this model yields a strong LP relaxation. It implies, in particular, general classes of facet defining inequalities for the standard model.

**1998 ACM Subject Classification** G.2.3 Applications in Discrete Mathematics

**Keywords and phrases** combinatorial optimization, polyhedral combinatorics, line planning

**Digital Object Identifier** 10.4230/OASIcs.ATMOS.2013.68

## 1 Introduction

*Line planning* is an important strategic planning problem in public transport. The task is to find a set of lines and frequencies such that a given demand can be transported. There are usually two main objectives: minimizing the travel times of the passengers and minimizing the line operating costs.

Since the late nineteen-nineties, the line planning literature has developed a variety of integer programming approaches that capture different aspects, see Schöbel [15] for an overview. Bussieck, Kreuzer, and Zimmermann [8] (see also the thesis of Bussieck [7]) propose an integer programming model to maximize the number of direct travelers. Operating costs are discussed in the articles of Claessens, van Dijk, and Zwaneveld [9] and Goossens, van Hoesel, and Kroon [11, 12]. Schöbel and Scholl [16] and Borndörfer and Karbstein [3] focus on the number of transfers and the number of direct travelers, respectively, and further integrate line planning and passenger routing in their models. Borndörfer, Grötschel, and Pfetsch [2] also propose an integrated line planning and passenger routing model that allows to generate lines dynamically.

All these models employ some type of *capacity* or *frequency demand constraints* in order to cover a given demand. In this paper we propose a concept to strengthen such constraints by means of a novel extended formulation. The idea is to enumerate the set of possible *configurations* of line frequencies for each capacity constraint. We show that such an extended formulation implies general facet defining inequalities for the standard model. We remark that configuration models have also been used successfully in railway vehicle rotation planning [4] and railway track allocation applications [5].

* Supported by the DFG Research Center MATHEON “Mathematics for key technologies”.

13th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS’13).
Editors: Daniele Frigioni, Sebastian Stiller; pp. 68–79
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

## 2 Problem Description

We consider the following basic *line planning problem.* We have an undirected graph $G = (V, E)$ representing the transportation network, and a set $\mathcal{L} = \{l_1, \dots, l_n\}$, $n \in \mathbb{N}$, of lines, where every line $l_i$ is a path in $G$. Denote by $\mathcal{L}(e) := \{l \in \mathcal{L} : e \in l\}$ the set of lines on edge $e \in E$. Furthermore, we are given an ordered set of frequencies $\mathcal{F} = \{f_1, \dots, f_k\} \subseteq \mathbb{N}$, such that $0 < f_1 < \dots < f_k$, $k \in \mathbb{N}$, and costs $c_{l,f}$ for operating line $l \in \mathcal{L}$ at frequency $f \in \mathcal{F}$. Finally, each edge $e \in E$ in the network bears a positive frequency demand $F(e)$ giving the number of line operations that are necessary to cover the demand on this edge.

A *line plan* $(\bar{\mathcal{L}}, \bar{f})$ consists of a subset $\bar{\mathcal{L}} \subseteq \mathcal{L}$ of lines and an assignment $\bar{f} : \bar{\mathcal{L}} \to \mathcal{F}$ of frequencies to these lines. A line plan is *feasible* if the frequencies of the lines satisfy the given frequency demand $F(e)$ for each edge $e \in E$, i.e., if

$$\sum_{l \in \bar{\mathcal{L}}(e)} \bar{f}(l) \geq F(e) \text{ for all } e \in E. \tag{1}$$

We define the cost of a line plan $(\bar{\mathcal{L}}, \bar{f})$ as $c(\bar{\mathcal{L}}, \bar{f}) = \sum_{l \in \bar{\mathcal{L}}} c_{l,\bar{f}(l)}$. The *line planning problem* is to find a feasible line plan of minimal cost.

### 2.1 Standard Model

The common way to formulate the line planning problem uses binary variables $x_{l,f}$ indicating whether line $l \in \mathcal{L}$ is operated at frequency $f \in \mathcal{F}$, cf. the references listed in the introduction. In our case, this results in the following *standard model*:

$$\begin{array}{llrll}
\text{(SLP)} & \min & \displaystyle\sum_{l \in \mathcal{L}} \sum_{f \in \mathcal{F}} c_{l,f} x_{l,f} & & \\
& \text{s.t.} & \displaystyle\sum_{l \in \mathcal{L}(e)} \sum_{f \in \mathcal{F}} f \cdot x_{l,f} \geq F(e) & \forall e \in E & (2) \\
& & \displaystyle\sum_{f \in \mathcal{F}} x_{l,f} \leq 1 & \forall l \in \mathcal{L} & (3) \\
& & x_{l,f} \in \{0,1\} & \forall l \in \mathcal{L}, \forall f \in \mathcal{F}. & (4)
\end{array}$$

Model (SLP) minimizes the cost of a line plan. The *frequency demand constraints* (2) ensure that the frequency demand is covered. The *assignment constraints* (3) ensure that every line operates at only one frequency. Hence, the solutions of (SLP) correspond to the feasible line plans.

### 2.2 Extended or Configuration Model

In the following, we give an extended formulation for (SLP) in order to tighten the LP-relaxation. The formulation is based on the observation that the frequency demand for an edge $e \in E$ can also be expressed by specifying the numbers $q_f$ of lines that are operated at frequency $f$, $f \in \mathcal{F}$, on edge $e$. We explain the idea using the example in Figure 1. The transportation network consists of two edges and three lines. Each line can be operated at frequency 2 or 8. The frequency demand on edge $\{u, v\}$ is 9. To cover this demand using at most three lines we need at least two lines with frequency 8 or one line with frequency 2 and one line with frequency 8. We call these feasible frequency combinations *configurations.* In this case the set of all possible configurations is $\bar{Q}(\{u, v\}) = \{(0, 2), (0, 3), (1, 1), (1, 2), (2, 1)\}$, where the first coordinate gives the number of lines with frequency 2 and the second coordinate gives the number of lines with frequency 8. The set of minimal configurations is $Q(\{u,v\}) = \{(0,2),(1,1)\}$, i.e., any line plan that matches one configuration covers the frequency demand of the edge $\{u,v\}$. Similarly, the minimal configurations for edge $\{v,w\}$ are $Q(\{v,w\}) = \{(0,1),(1,0)\}$. Any line plan that matches one configuration for each edge is a feasible line plan. A formal description is as follows.

▶ **Definition 1.** For $e \in E$ denote by

$$\bar{\mathcal{Q}}(e) := \Big\{ q = (q_{f_1},\dots,q_{f_k}) \in \mathbb{N}_0^k : \sum_{f\in\mathcal{F}} q_f \le |\mathcal{L}(e)|, \sum_{f\in\mathcal{F}} f\cdot q_f \ge F(e) \Big\}$$

the set of *(feasible) (frequency) configurations of* $e$ and by

$$\mathcal{Q}(e) := \big\{ q \in \bar{\mathcal{Q}}(e) : (q_{f_1},\dots,q_{f_i}-1,\dots,q_{f_k}) \notin \bar{\mathcal{Q}}(e) \quad \forall i = 1,\dots,k \big\}$$

the set of *minimal configurations of* $e$.

Let $q = \big(q(e)\big)_{e\in E}$ be some vector of minimal configurations, i.e., $q(e) \in \mathcal{Q}(e)$ for all $e \in E$, and $(\bar{\mathcal{L}},\bar{f})$ a line plan. If $(\bar{\mathcal{L}},\bar{f})$ satisfies the inequality $|\{l \in \bar{\mathcal{L}}(e) : \bar{f}(l) = f\}| \ge q(e)_f$ for all $e \in E$, $f \in \mathcal{F}$, then $(\bar{\mathcal{L}},\bar{f})$ is feasible. Conversely, if $(\bar{\mathcal{L}},\bar{f})$ is a feasible line plan, then $q_f = |l \in \bar{\mathcal{L}}(e) : \bar{f}(l) = f|$, $f \in \mathcal{F}$, is a feasible frequency configuration for each edge $e$. In other words, satisfying the frequency demand is equivalent to choosing a feasible configuration for each edge.

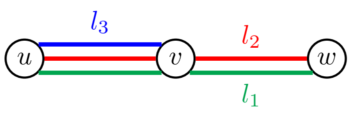
We extend (SLP) using binary variables $y_{e,q}$ that indicate for each edge $e \in E$ which configuration $q \in \mathcal{Q}$ is chosen. This results in the following extended formulation:

$$\begin{array}{llllr}
\text{(QLP)} & \min & \displaystyle\sum_{l\in\mathcal{L}}\sum_{f\in\mathcal{F}} c_{l,f}x_{l,f} & & \\
& \text{s.t.} & \displaystyle\sum_{l\in\mathcal{L}(e)} x_{l,f} \ge \sum_{q\in\mathcal{Q}(e)} q_f\cdot y_{e,q} & \forall e\in E, \forall f\in\mathcal{F} & (5)\\
& & \displaystyle\sum_{q\in\mathcal{Q}(e)} y_{e,q} = 1 & \forall e \in E & (6)\\
& & \displaystyle\sum_{f\in\mathcal{F}} x_{l,f} \le 1 & \forall l\in\mathcal{L} & (7)\\
& & x_{l,f} \in \{0,1\} & \forall l\in\mathcal{L}, \forall f\in\mathcal{F} & (8)\\
& & y_{e,q} \in \{0,1\} & \forall e\in E, \forall q\in\mathcal{Q}(e). & (9)
\end{array}$$

The *(extended) configuration model* (QLP) also minimizes the cost of a line plan. The *configuration assignment constraints* (6) ensure that exactly one configuration for each edge is chosen while the *coupling constraints* (5) guarantee that sufficient numbers of lines are operated at the frequencies of the chosen configurations.

▶ **Example 2.** Consider the line planning problem in Figure 1. We define the costs as $c_{l_1,f} = c_{l_2,f} = 2\cdot f$ and $c_{l_3,f} = f$. The standard model for this example reads as follows:

$$\begin{array}{lllllll}
\text{(SLP)} & \min & 4x_{l_1,2}+16x_{l_1,8} & +4x_{l_2,2}+16x_{l_2,8} & +2x_{l_3,2}+8x_{l_3,8} & & \\
& \text{s.t.} & 2x_{l_1,2}+8x_{l_1,8} & +2x_{l_2,2}+8x_{l_2,8} & +2x_{l_3,2}+8x_{l_3,8} & \ge 9 & \\
& & 2x_{l_1,2}+8x_{l_1,8} & +2x_{l_2,2}+8x_{l_2,8} & & \ge 1 & \\
& & x_{l_1,2}\ +x_{l_1,8} & & & \le 1 & \\
& & & +x_{l_2,2}\ +x_{l_2,8} & & \le 1 & \\
& & & & +x_{l_3,2}\ +x_{l_3,8} & \le 1 & \\
& & & & x_{l_i,f} & \in \{0,1\}. &
\end{array}$$



$\mathcal{F} = \{2, 8\}$

| $e$ | $e_1 = \{u, v\}$ | $e_2 = \{v, w\}$ |
|---|---|---|
| $F(e)$ | 9 | 1 |
| $\mathcal{L}(e)$ | $\{l_1, l_2, l_3\}$ | $\{l_1, l_2\}$ |
| $\mathcal{Q}(e)$ | $\{(0, 2), (1, 1)\}$ | $\{(0, 1), (1, 0)\}$ |

**Figure 1** An instance of the line planning problem. *Left*: Transportation network consisting of two edges and three lines. *Right*: The given set of frequencies, frequency demands, and the minimal frequency configurations.

The configuration model for this example is:

$$
\begin{array}{llcccccccccccl}
\text{(QLP)} & \min & 4x_{l_1,2} & +16x_{l_1,8} & +4x_{l_2,2} & +16x_{l_2,8} & +2x_{l_3,2} & +8x_{l_3,8} & & & & & & \\
 & \text{s.t.} & x_{l_1,2} & & +x_{l_2,2} & & +x_{l_3,2} & & & -y_{e_1,q_2} & & & \geq 0 & \\
 & & & +x_{l_1,8} & & +x_{l_2,8} & & +x_{l_3,8} & -2y_{e_1,q_1} & -y_{e_1,q_2} & & & \geq 0 & \\
 & & x_{l_1,2} & & +x_{l_2,2} & & & & & & & -y_{e_2,q_2} & \geq 0 & \\
 & & & +x_{l_1,8} & & +x_{l_2,8} & & & & & -y_{e_2,q_1} & & \geq 0 & \\
 & & x_{l_1,2} & +x_{l_1,8} & & & & & & & & & \leq 1 & \\
 & & & & +x_{l_2,2} & +x_{l_2,8} & & & & & & & \leq 1 & \\
 & & & & & & +x_{l_3,2} & +x_{l_3,8} & & & & & \leq 1 & \\
 & & & & & & & & +y_{e_1,q_1} & +y_{e_1,q_2} & & & = 1 & \\
 & & & & & & & & & & +y_{e_2,q_1} & +y_{e_2,q_2} & = 1 & \\
 & & & & & & & & & & & x_{l_i,f} & \in \{0, 1\} & \\
 & & & & & & & & & & & y_{e,q} & \in \{0, 1\}. &
\end{array}
$$

## 3 Comparison of the Models

In this section we compare the standard and the extended configuration model for the line planning problem. We need some further notation. For an integer program (IP) $= \min\{c^T x : Ax \geq b, x \in \mathbb{Z}^n\}$ we denote by $\mathrm{P_{IP}}$ (IP) the polyhedron defined by the convex hull of all feasible solutions of (IP) and by $\mathrm{P_{LP}}$ (IP) the set of feasible solutions of the LP relaxation of (IP), i.e., $\mathrm{P_{IP}}$ (IP) $= \operatorname{conv}\{x \in \mathbb{Z}^n : Ax \geq b\}$ and $\mathrm{P_{LP}}$ (IP) $= \{x \in \mathbb{R}^n : Ax \geq b\}$. For a polyhedron $P = \{(x, y) \in \mathbb{R}^{n+m} : Ax + By \geq b\}$ denote by $P|_x := \{x \in \mathbb{R}^n : \exists y \in \mathbb{R}^m \text{ s.t. } (x, y) \in P\}$ the projection of $P$ onto the space of $x$-variables.

Using this notation, we can state that solving (QLP) is equivalent to solving (SLP):

▶ **Lemma 3.** (QLP) *provides an extended formulation for* (SLP), *i.e.,*

$$\mathrm{P_{IP}}(\text{QLP})\,|_x = \mathrm{P_{IP}}(\text{SLP})\,.$$

For the LP relaxations, however, the following holds:

▶ **Theorem 4.** *The LP relaxation of* $\mathrm{P_{IP}}(\text{QLP})\,|_x$ *is tighter than the LP relaxation of* $\mathrm{P_{IP}}(\text{SLP})$, *i.e.,*

$$\mathrm{P_{LP}}(\text{QLP})\,|_x \subseteq \mathrm{P_{LP}}(\text{SLP})\,.$$

**Proof.** Let $(\bar{x}, \bar{y}) \in \mathrm{P_{LP}}(\text{QLP})$. Obviously, $\bar{x}$ satisfies (3) and (4). We further get

$$\sum_{l \in \mathcal{L}(e)} \sum_{f \in \mathcal{F}} f \cdot \bar{x}_{l,f} \overset{(5)}{\geq} \sum_{f \in \mathcal{F}} \Big( f \cdot \sum_{q \in \mathcal{Q}(e)} q_f \cdot \bar{y}_{e,q} \Big) = \underbrace{\sum_{q \in \mathcal{Q}(e)} \bar{y}_{e,q}}_{\overset{(6)}{=} 1} \cdot \underbrace{\sum_{f \in \mathcal{F}} f \cdot q_f}_{\geq F(e) \forall q \in \mathcal{Q}(e)} \geq F(e).$$

Hence, $\bar{x}$ satisfies (2) as well and is contained in $\mathrm{P_{LP}}(\text{SLP})$. ◀

The converse, i.e., $\mathrm{P_{LP}}(\mathrm{SLP}) \subseteq \mathrm{P_{LP}}(\mathrm{QLP})\,|_x$, does not hold in general, indeed, the ratio of the optimal objectives of the two LP relaxations can be arbitrarily large.

▶ **Example 5.** Consider an instance of the line planning problem involving only one edge $E = \{e\}$, one line $\mathcal{L}(e) = \{l\}$, a frequency demand $F(e) = 6$, and one frequency $\mathcal{F} = \{M\}$ such that $M > 6$ with cost function $c_{l,M} = M$. The only minimal configuration for $e$ is $q = (1)$.

$$\begin{aligned} \mathrm{QLP_{LP}}:\quad \min \quad & M \cdot x_{l,M} \\ \text{s.t. } x_{l,M} - y_q &\geq 0 \\ y_q &= 1 \\ y_q, x_{l,M} &\geq 0 \end{aligned} \qquad \begin{aligned} \mathrm{SLP_{LP}}:\quad \min \quad & M \cdot x_{l,M} \\ \text{s.t. } M \cdot x_{l,M} &\geq 6 \\ \\ x_{l,M} &\geq 0 \end{aligned}$$

Obviously, $x_{l,M} = 1$ is the only and hence optimal solution to $\mathrm{QLP_{LP}}$ with objective value $M$ and $x_{l,M} = \frac{6}{M}$ is an optimal solution to $\mathrm{SLP_{LP}}$ with objective value 6.

In the following subsections we show that the LP relaxation of the configuration model implies general classes of facet defining inequalities for the line planning polytope $\mathrm{P_{IP}}(\mathrm{SLP})$ that are discussed in the literature.

## 3.1 Band Inequalities

In this section we analyze band inequalities, which were introduced by Stoer and Dahl [19] and are closely related to the knapsack cover inequalities, see Wolsey [21].

▶ **Definition 6.** Let $e \in E$.

- A *band* $f_{\mathcal{B}} : \mathcal{L}(e) \to \mathcal{F} \cup \{0\}$ assigns to each line containing $e$ a frequency or 0. We call $f_{\mathcal{B}}$ a *valid band of* $e$ if
$$\sum_{l \in \mathcal{L}(e)} f_{\mathcal{B}}(l) < F(e).$$
- We call the band $f_{\mathcal{B}}$ *maximal* if $f_{\mathcal{B}}$ is valid and there is no valid band $f_{\mathcal{B}'}$ with $f_{\mathcal{B}}(l) \leq f_{\mathcal{B}'}(l)$ for every line $l \in \mathcal{L}(e)$ and $f_{\mathcal{B}}(l) < f_{\mathcal{B}'}(l)$ for at least one line $l \in \mathcal{L}(e)$.
- We call the band $f_{\mathcal{B}}$ *symmetric* if $f_{\mathcal{B}}(l) = f$ for all $l \in \mathcal{L}(e)$ and for some $f \in \mathcal{F}$.

Applying the results of Stoer and Dahl [19] yields

▶ **Proposition 7.** *Let $f_{\mathcal{B}}$ be a valid band of $e \in E$, then*

$$\sum_{l \in \mathcal{L}(e)} \sum_{\substack{f \in \mathcal{F} \\ f > f_{\mathcal{B}}(l)}} x_{l,f} \geq 1 \tag{10}$$

*is a valid inequality for* $\mathrm{P_{IP}}(\mathrm{SLP})$.

The simplest example is the case $f_{\mathcal{B}}(l) \equiv 0$, which states that one must operate at least one line on every edge, i.e., one has to cover the demand.

▶ **Proposition 8.** *The* set cover *inequality*

$$\sum_{l \in \mathcal{L}(e)} \sum_{f \in \mathcal{F}} x_{l,f} \geq 1 \tag{11}$$

*is valid for* $\mathrm{P_{IP}}(\mathrm{SLP})$ *for all* $e \in E$.

The set cover inequalities (11) do not hold in general for the LP relaxation of the standard model, compare with Example 5. Note that they are symmetric band inequalities.

Maximal band inequalities often define facets of the single edge relaxation of the line planning polytope [13]. The symmetric ones are implied by the configuration model.

▶ **Theorem 9.** *The LP relaxation of the configuration model implies all band inequalities* (10) *that are induced by a valid symmetric band.*

**Proof.** Assume $f_{\mathcal{B}}$ is a valid symmetric band of some edge $e$ with $f_{\mathcal{B}}(l) = \tilde{f}$ for all $l \in \mathcal{L}(e)$ and for some $\tilde{f} \in \mathcal{F}$, $\tilde{f} < f_k$. Thus $\sum_{l \in \mathcal{L}(e)} f_{\mathcal{B}}(l) = |\mathcal{L}(e)| \cdot \tilde{f} < F(e)$. Hence, in every minimal configuration $q \in \mathcal{Q}(e)$ there is a frequency $f > \tilde{f}$ such that $q_f \geq 1$. Starting from (5), we get:

$$\begin{aligned}
& & \sum_{l \in \mathcal{L}(e)} x_{l,f} &\geq \sum_{q \in \mathcal{Q}(e)} q_f \cdot y_q \qquad \forall f \in \mathcal{F} \\
&\Rightarrow & \sum_{\substack{f \in \mathcal{F} \\ f > \tilde{f}}} \sum_{l \in \mathcal{L}(e)} x_{l,f} &\geq \sum_{\substack{f \in \mathcal{F} \\ f > \tilde{f}}} \sum_{q \in \mathcal{Q}(e)} q_f \cdot y_q \\
&\Leftrightarrow & \sum_{l \in \mathcal{L}(e)} \sum_{\substack{f \in \mathcal{F} \\ f > \tilde{f}}} x_{l,f} &\geq \sum_{q \in \mathcal{Q}(e)} y_q \cdot \underbrace{\sum_{\substack{f \in \mathcal{F} \\ f > \tilde{f}}} q_f}_{\geq 1} \\
& & &\geq \sum_{q \in \mathcal{Q}(e)} y_q = 1.
\end{aligned}$$

◀

The same does not hold for the standard model as the following example shows.

▶ **Example 10** (Example 2 continued)**.** A valid symmetric band for edge $e_1$ in Figure 1 is given by $f_{\mathcal{B}}(l) = 2$ for all $l \in \mathcal{L}(e_1)$. The corresponding band inequality

$$x_{l_1,8} + x_{l_2,8} + x_{l_3,8} \geq 1 \tag{12}$$

is violated by $\tilde{x} \in \mathrm{P}_{\mathrm{LP}}(\mathrm{SLP})$, where $\tilde{x}_{l_2,8} = \frac{7}{8}$, $\tilde{x}_{l_3,2} = 1$, and $\tilde{x}_{l,f} = 0$ otherwise. One can show that (12) is facet-defining for $\mathrm{P}_{\mathrm{IP}}(\mathrm{SLP})$ in this example.

## 3.2 MIR Inequalities

We study in this section the mixed integer rounding (MIR) inequalities and their connection to the configuration model. MIR inequalities can be derived from the basic MIR inequality as defined by Wolsey [22], see also Raack [14].

▶ **Lemma 11** (Wolsey [22])**.** *Let* $Q_I := \{(x,y) \in \mathbb{Z} \times \mathbb{R} : x + y \geq \beta, y \geq 0\}$. *The basic MIR inequality*

$$rx + y \geq r\lceil \beta \rceil$$

*with* $r := r(\beta) = \beta - \lfloor \beta \rfloor$ *is valid for* $Q_I$ *and defines a facet of* $\mathrm{conv}(Q_I)$ *if* $r > 0$.

We use mixed integer rounding to strengthen the demand inequalities (2).

▶ **Proposition 12.** *Let $\lambda \in \mathbb{R}_+$, $e \in E$, and define $r = \lambda F(e) - \lfloor \lambda F(e) \rfloor$ and $r_f = \lambda f - \lfloor \lambda f \rfloor$. The MIR inequality*

$$\sum_{l \in \mathcal{L}(e)} \sum_{f \in \mathcal{F}} (r \lfloor \lambda f \rfloor + \min(r_f, r)) x_{l,f} \geq r \lceil \lambda F(e) \rceil \tag{13}$$

*induced by the demand inequality* (2) *scaled by $\lambda$ is valid for* (SLP)*.*

**Proof.** Scaling inequality (2) by $\lambda > 0$ yields

$$\begin{aligned}
\lambda \cdot F(e) \leq \lambda \cdot \sum_{l \in \mathcal{L}(e)} \sum_{f \in \mathcal{F}} f \cdot x_{l,f} &= \sum_{l \in \mathcal{L}(e)} \sum_{\substack{f \in \mathcal{F} \\ r_f < r}} \lambda \cdot f \cdot x_{l,f} + \sum_{l \in \mathcal{L}(e)} \sum_{\substack{f \in \mathcal{F} \\ r_f \geq r}} \lambda \cdot f \cdot x_{l,f} \\
&\leq \sum_{l \in \mathcal{L}(e)} \sum_{\substack{f \in \mathcal{F} \\ r_f < r}} (\lfloor \lambda \cdot f \rfloor + r_f) \cdot x_{l,f} + \sum_{l \in \mathcal{L}(e)} \sum_{\substack{f \in \mathcal{F} \\ r_f \geq r}} (\lfloor \lambda \cdot f \rfloor + 1) \cdot x_{l,f} \\
&= \underbrace{\sum_{l \in \mathcal{L}(e)} \sum_{\substack{f \in \mathcal{F} \\ r_f < r}} r_f \cdot x_{l,f}}_{\geq 0} + \underbrace{\sum_{l \in \mathcal{L}(e)} \sum_{f \in \mathcal{F}} \lfloor \lambda \cdot f \rfloor \cdot x_{l,f} + \sum_{l \in \mathcal{L}(e)} \sum_{\substack{f \in \mathcal{F} \\ r_f \geq r}} x_{l,f}}_{\in \mathbb{Z}}.
\end{aligned}$$

Applying Lemma 11 yields

$$\begin{aligned}
r \cdot \lceil \lambda \cdot F(e) \rceil &\leq \sum_{l \in \mathcal{L}(e)} \sum_{\substack{f \in \mathcal{F} \\ r_f < r}} r_f \cdot x_{l,f} + r \cdot \Big( \sum_{l \in \mathcal{L}(e)} \sum_{f \in \mathcal{F}} \lfloor \lambda \cdot f \rfloor \cdot x_{l,f} + \sum_{l \in \mathcal{L}(e)} \sum_{\substack{f \in \mathcal{F} \\ r_f \geq r}} x_{l,f} \Big) \\
&= \sum_{l \in \mathcal{L}(e)} \sum_{f \in \mathcal{F}} (r \cdot \lfloor \lambda f \rfloor + \min(r_f, r)) \cdot x_{l,f}.
\end{aligned}$$

◀

Notice that $\lambda \in \mathbb{R}_+$ only produces a non-trivial MIR inequality (13) if $r = \lambda F(e) - \lfloor \lambda F(e) \rfloor \neq 0$. Dash, Günlük and Lodi [10] analyze for which $\lambda$ the MIR inequality (13) is non-redundant.

▶ **Proposition 13** (Dash, Günlük and Lodi [10])**.** *Each non-redundant MIR inequality* (13) *is defined by $\lambda \in (0, 1)$, where $\lambda$ is a rational number with denominator equal to some $f \in \mathcal{F}$.*

Again, we can show that these inequalities are implied by the LP relaxation of the configuration model. The proof is based on the following lemma, a configuration version of Proposition 12.

▶ **Lemma 14.** *For $e \in E$, $q \in \mathcal{Q}(e)$, and $\lambda \in (0, 1)$, it holds*

$$\sum_{f \in \mathcal{F}} (r \cdot \lfloor \lambda f \rfloor + \min(r_f, r)) q_f \geq r \cdot \lceil \lambda \cdot F(e) \rceil,$$

*where $r = \lambda F(e) - \lfloor \lambda F(e) \rfloor$ and $r_f = \lambda f - \lfloor \lambda f \rfloor$.*

**Proof.** $q \in \mathcal{Q}(e)$ implies $\sum\limits_{f\in\mathcal{F}} f \cdot q_f \geq F(e)$ and hence we get for $\lambda \in (0,1)$

$$\begin{aligned}
\lambda \cdot F(e) \leq \lambda \cdot \sum_{f\in\mathcal{F}} f \cdot q_f &= \sum_{\substack{f\in\mathcal{F}\\ r_f<r}} \lambda \cdot f \cdot q_f + \sum_{\substack{f\in\mathcal{F}\\ r_f\geq r}} \lambda \cdot f \cdot q_f \\
&\leq \sum_{\substack{f\in\mathcal{F}\\ r_f<r}} (\lfloor \lambda \cdot f \rfloor + r_f) \cdot q_f + \sum_{\substack{f\in\mathcal{F}\\ r_f\geq r}} (\lfloor \lambda \cdot f \rfloor + 1) \cdot q_f \\
&= \underbrace{\sum_{\substack{f\in\mathcal{F}\\ r_f<r}} r_f \cdot q_f}_{\geq 0} + \underbrace{\sum_{f\in\mathcal{F}} \lfloor \lambda \cdot f \rfloor \cdot q_f + \sum_{\substack{f\in\mathcal{F}\\ r_f\geq r}} q_f}_{\in\mathbb{Z}}.
\end{aligned}$$

Applying Lemma 11 yields

$$\begin{aligned}
r \cdot \lceil \lambda \cdot F(e) \rceil &\leq \sum_{\substack{f\in\mathcal{F}\\ r_f<r}} r_f \cdot q_f + r \cdot \Bigg( \sum_{f\in\mathcal{F}} \lfloor \lambda \cdot f \rfloor \cdot q_f + \sum_{\substack{f\in\mathcal{F}\\ r_f\geq r}} q_f \Bigg) \\
&= \sum_{f\in\mathcal{F}} (r \cdot \lfloor \lambda f \rfloor + \min(r_f, r)) \cdot q_f.
\end{aligned}$$

◀

▶ **Theorem 15.** *Let $\lambda \in (0,1)$, $e \in E$, $r = \lambda F(e) - \lfloor \lambda F(e) \rfloor$ and $r_f = \lambda f - \lfloor \lambda f \rfloor$. Then the MIR inequality*

$$\sum_{l\in\mathcal{L}(e)} \sum_{f\in\mathcal{F}} (r \lfloor \lambda f \rfloor + \min(r_f, r)) x_{l,f} \geq r \lceil \lambda F(e) \rceil$$

*is implied by the LP relaxation of the configuration model, i.e., the MIR inequalities* (13) *are valid for* $\mathrm{P_{LP}}(\mathrm{QLP})\,|_x$.

**Proof.** Let $(x,y) \in \mathrm{P_{LP}}(\mathrm{QLP})$. Then by (5)

$$\sum_{l\in\mathcal{L}(e)} x_{l,f} \geq \sum_{q\in\mathcal{Q}(e)} q_f \cdot y_q \qquad \forall f \in \mathcal{F}.$$

Scaling this inequality by $\lambda_f^r := r \cdot \lfloor \lambda f \rfloor + \min(r_f, r)$ yields

$$\begin{aligned}
&\sum_{l\in\mathcal{L}(e)} \lambda_f^r \cdot x_{l,f} \geq \sum_{q\in\mathcal{Q}(e)} \lambda_f^r \cdot q_f \cdot y_q \qquad \forall f \in \mathcal{F} \\
\Rightarrow &\sum_{f\in\mathcal{F}} \sum_{l\in\mathcal{L}(e)} \lambda_f^r \cdot x_{l,f} \geq \sum_{f\in\mathcal{F}} \sum_{q\in\mathcal{Q}(e)} \lambda_f^r \cdot q_f \cdot y_q \\
\Leftrightarrow &\sum_{l\in\mathcal{L}(e)} \sum_{f\in\mathcal{F}} \lambda_f^r \cdot x_{l,f} \geq \sum_{q\in\mathcal{Q}(e)} \sum_{f\in\mathcal{F}} \lambda_f^r \cdot q_f \cdot y_q \\
&\overset{(*)}{\geq} \sum_{q\in\mathcal{Q}(e)} r \cdot \lceil \lambda \cdot F(e) \rceil \cdot y_q \\
&= r \cdot \lceil \lambda \cdot F(e) \rceil \cdot \sum_{q\in\mathcal{Q}(e)} y_q \\
&\overset{(6)}{=} r \cdot \lceil \lambda \cdot F(e) \rceil.
\end{aligned}$$

$(*)$ apply Lemma 14 here. ◀

**Table 1** Statistics on the line planning instances. The columns list the instance name, the number of edges of the preprocessed transportation network, the number of lines, the number of variables for lines and frequencies, and the number of configuration variables in the configuration model and in the mixed model.

| | | | (SLP)/(SLP$^+$) | | (SLP$^Q$) | | (QLP) | |
|---|---|---|---|---|---|---|---|---|
| name | $\lvert E\rvert$ | $\lvert\mathcal{L}\rvert$ | #vars | #cons | #vars | #cons | #vars | #cons |
| China1 | 27 | 474 | 2 793 | 499 / 620 | 3 732 | 654 | 41 196 | 661 |
| China2 | 27 | 4 871 | 29 170 | 4 896 / 5 016 | 36 757 | 5 058 | 67 575 | 5 058 |
| China3 | 27 | 19 355 | 116 074 | 19 380 /19 500 | 145 736 | 19 542 | 154 479 | 19 542 |
| Dutch1 | 30 | 402 | 1 544 | 424 / 502 | 1 760 | 580 | 1 760 | 580 |
| Dutch2 | 30 | 2 679 | 11 779 | 2 701 / 2 779 | 11 997 | 2 859 | 11 997 | 2 859 |
| Dutch3 | 30 | 7 302 | 33 988 | 7 324 / 7 402 | 34 206 | 7 482 | 34 206 | 7 482 |
| SiouxFalls1 | 37 | 866 | 5 188 | 902 / 1 113 | 6 680 | 1 117 | 753 840 | 1 124 |
| SiouxFalls2 | 37 | 9 397 | 56 374 | 9 433 / 9 644 | 73 531 | 9 648 | 902 703 | 9 655 |
| SiouxFalls3 | 37 | 15 365 | 92 182 | 15 401 /15 612 | 117 711 | 15 616 | 938 511 | 15 623 |
| Potsdam1998b | 351 | 1 907 | 10 765 | 1 998 / 2 679 | 13 795 | 3 969 | 38 637 | 4 114 |
| Potsdam1998c | 351 | 4342 | 25 306 | 4 431 / 5 112 | 32 037 | 6 484 | 53 184 | 6 549 |
| Potsdam2010 | 517 | 3433 | 9 535 | 3 109 / 3 584 | 11 524 | 4 986 | 11 524 | 4 986 |
| Chicago | 1 028 | 23 109 | 131 915 | 24 066 /28 297 | 165 083 | 30 229 | 2 503 163 | 30 285 |

Again, we can give an example where a MIR inequality is not valid for the LP relaxation of the standard model.

▶ **Example 16** (Example 2 continued)**.** Let $\lambda = \frac{1}{8}$, then the MIR inequality for edge $e_1$

$$x_{l_1,2} + x_{l_1,8} + x_{l_2,2} + x_{l_2,8} + x_{l_3,2} + x_{l_3,8} \geq 2 \tag{14}$$

is violated by $\tilde{x} \in \mathrm{P_{LP}(SLP)}$, where $\tilde{x}_{l_2,8} = \frac{7}{8}$, $\tilde{x}_{l_3,2} = 1$, and $\tilde{x}_{l,f} = 0$ otherwise. It can be verified that (14) is even facet-defining for $\mathrm{P_{IP}(SLP)}$ in this example.

## 4 Computational Results

We have implemented the configuration approach to provide a computational evaluation of the strength of the extended formulation (QLP). We compare it with the standard model (SLP) and with two additional models (SLP$^+$) and (SLP$^Q$). Model (SLP$^+$) is obtained by adding the set cover, symmetric band, and MIR inequalities for all edges to the standard model (SLP). Model (SLP$^Q$) has been developed to cut down on the number of configuration variables, which can explode for large instances. This model is situated between (SLP$^+$) and (QLP) and constructed as follows. We order the edges with respect to an increasing number of minimal configurations and generate the configuration variables and the associated constraints iteratively as long as the number of generated configuration variables does not exceed 25% of the number of variables for lines and frequencies. For the remaining edges we use the set cover, symmetric band, and MIR inequalities.

Our test set consists of five transportation networks that we denote as China, Dutch, SiouxFalls, Chicago, and Potsdam. The instances SiouxFalls and Chicago use the graph and the demand of the street network with the same name from the Transportation Network Test Problems Library of Bar-Gera [20]. Instances China, Dutch, and Potsdam correspond to public transportation networks. The Dutch network was introduced by Bussieck in the context of line planning [6]. The China instance is artificial; we constructed it as a showcase example, connecting the twenty biggest cities in China by the 2009 high speed

**Table 2** Statistics on the computations for the models (SLP), (SLP$^+$), (SLP$^Q$), and (QLP). The columns list the instance name, model, computation time, number of branching nodes, the integrality gap, the primal bound, the dual bound, and the dual bound after solving the root node.

| name | model | time | nodes | gap | primal | dual | root dual |
|---|---|---|---|---|---|---|---|
| China1 | (SLP) | 1h | 1524169 | 1.22 % | 236631,2 | 233772.3 | 233566.3 |
| | (SLP$^+$) | 1h | 808186 | 0.37 % | 235873.4 | 235006.5 | 234772.3 |
| | (SLP$^Q$) | 1h | 1147588 | 0.21 % | 235531.2 | 235038.6 | 234828.6 |
| | (QLP) | 1h | 31204 | 0.49 % | 236149.0 | 235005.2 | 234878.3 |
| China2 | (SLP) | 1h | 154009 | 2.47 % | 238187.4 | 232436.5 | 232294.8 |
| | (SLP$^+$) | 1h | 24751 | 1.42 % | 237333.4 | 234011.1 | 233860.8 |
| | (SLP$^Q$) | 1h | 21388 | 0.50 % | 235249.0 | 234076.8 | 233890.2 |
| | (QLP) | 1h | 13872 | 0.63 % | 235549.0 | 234071.3 | 233891.2 |
| China3 | (SLP) | 1h | 21078 | 3.78 % | 241046.0 | 232271.9 | 232203.5 |
| | (SLP$^+$) | 1h | 2214 | 0.99 % | 236067.0 | 233760.1 | 233735.5 |
| | (SLP$^Q$) | 1h | 3880 | 0.88 % | 235925.8 | 233862.9 | 233778.1 |
| | (QLP) | 1h | 3914 | 1.20 % | 236639.6 | 233844.8 | 233778.1 |
| Dutch1 | (SLP) | 1h | 7427826 | 1.03 % | 59000.0 | 58400.2 | 58227.4 |
| | (SLP$^+$) | 3.81s | 1301 | 0.00 % | 59000.0 | 59000.0 | 58841.7 |
| | (SLP$^Q$) | 0.98s | 23 | 0.00 % | 59000.0 | 59000.0 | 58868.6 |
| | (QLP) | 0.99s | 23 | 0.00 % | 59000.0 | 59000.0 | 58868.6 |
| Dutch2 | (SLP) | 1h | 609931 | 12.76 % | 59300.0 | 52587.5 | 52492.3 |
| | (SLP$^+$) | 1934.67s | 352128 | 0.00 % | 58600.0 | 58600.0 | 58392.2 |
| | (SLP$^Q$) | 45.62s | 6407 | 0.00 % | 58600.0 | 58600.0 | 58435.7 |
| | (QLP) | 45.62s | 6407 | 0.00 % | 58600.0 | 58600.0 | 58435.7 |
| Dutch3 | (SLP) | 1h | 87746 | 14.64 % | 59700.0 | 52075.0 | 52022.2 |
| | (SLP$^+$) | 1h | 168915 | 0.38 % | 58600.0 | 58376.6 | 58356.3 |
| | (SLP$^Q$) | 77.15s | 1915 | 0.00 % | 58500.0 | 58500.0 | 58372.9 |
| | (QLP) | 76.64s | 1915 | 0.00 % | 58500.0 | 58500.0 | 58372.9 |
| SiouxFalls1 | (SLP) | 1029.24s | 1115540 | 0.00 % | 2409.8 | 2409.8 | 2352.6 |
| | (SLP$^+$) | 270.45s | 125157 | 0.00 % | 2409.8 | 2409.8 | 2365.0 |
| | (SLP$^Q$) | 177.8s | 51099 | 0.00 % | 2409.8 | 2409.8 | 2357.2 |
| | (QLP) | 1h | 0 | infinite | - | - | - |
| SiouxFalls2 | (SLP) | 1h | 11664 | 26.07 % | 1815.3 | 1439.9 | 1439.9 |
| | (SLP$^+$) | 1h | 44565 | 3.48 % | 1704.2 | 1647.0 | 1647.0 |
| | (SLP$^Q$) | 1h | 19324 | 3.48 % | 1704.2 | 1647.0 | 1647.0 |
| | (QLP) | 1h | 0 | infinite | - | - | - |
| SiouxFalls3 | (SLP) | 1h | 27994 | 23.89 % | 1527.8 | 1233.2 | 1233.2 |
| | (SLP$^+$) | 1h | 6452 | 4.13 % | 1420.7 | 1364.4 | 1363.9 |
| | (SLP$^Q$) | 1h | 7569 | 3.83 % | 1416.3 | 1364.1 | 1363.9 |
| | (QLP) | 1h | 0 | infinite | - | - | - |
| Potsdam1998b | (SLP) | 1h | 233518 | 3.74 % | 36688.3 | 35365.0 | 35124.2 |
| | (SLP$^+$) | 1h | 123701 | 0.77 % | 36167.3 | 35891.0 | 35735.2 |
| | (SLP$^Q$) | 1h | 237661 | 0.36 % | 36067.0 | 35936.1 | 35770.6 |
| | (QLP) | 1h | 124082 | 0.14 % | 36067.0 | 36018.0 | 35850.4 |
| Potsdam1998c | (SLP) | 1h | 105062 | 4.47 % | 36617.1 | 35051.8 | 34896.9 |
| | (SLP$^+$) | 1h | 38634 | 1.69 % | 36243.5 | 35641.9 | 35510.1 |
| | (SLP$^Q$) | 1h | 63336 | 0.56 % | 35891.9 | 35690.7 | 35575.8 |
| | (QLP) | 1h | 11681 | 7.49 % | 38345.8 | 35675.3 | 35521.8 |
| Potsdam2010 | (SLP) | 2.47s | 1 | 0.00 % | 11066.6 | 11066.6 | 11066.6 |
| | (SLP$^+$) | 4.93s | 8 | 0.00 % | 11066.6 | 11066.6 | 11011.8 |
| | (SLP$^Q$) | 6.31s | 7 | 0.00 % | 11066.6 | 11066.6 | 11046.9 |
| | (QLP) | 6.25s | 7 | 0.00 % | 11066.6 | 11066.6 | 11046.9 |
| Chicago | (SLP) | 1h | 2002 | 5.88 % | 22990.6 | 21713.3 | 21666.6 |
| | (SLP$^+$) | 1h | 553 | 2.79 % | 22327.2 | 21722.2 | 21685.3 |
| | (SLP$^Q$) | 1h | 319 | 5.73 % | 22948.1 | 21705.0 | 21689.4 |
| | (QLP) | 1h | 0 | infinite | - | - | - |

train network. The Potsdam instances are real multi-modal public transportation networks for 1998 and 2009.

We constructed a line pool by generating for each pair of terminals all lines that satisfy a certain length restriction. To be more precise, the number of edges of a line between two terminals $s$ and $t$ must be less than or equal to $k$ times the number of edges of the shortest path between $s$ and $t$. For each network, we increased $k$ in three steps to produce three instances with different line pool sizes. For Dutch and China instance number 3 contains all lines, i.e., all paths that are possible in the network. The Potsdam2010 instance arose within a project with the Verkehr in Potsdam GmbH (ViP) [18] to optimize the 2010 line plan [1]. The line pool contains all possible lines that fulfill the ViP requirements.

For all instances the lines can be operated at frequencies 3, 6, 9, 18, 36, and 72. This corresponds to a cycle time of 60, 30, 20, 10, 5, and 2.5 minutes in a time horizon of 3 hours. We set the line cost to be proportional to the line length and the frequency plus a fixed cost term that is used to reduce the number of lines. The costs and the capacities of the lines depend on the mode of transportation (e.g., bus, tram). In the instances each edge is associated with exactly one mode, i.e., all lines on an edge have the same capacity, see Karbstein [13] for more details. Hence, we can express capacities in terms of frequency demands. Table 1 lists some statistics about the test instances. The second and third columns give the number of edges and lines in the transportation network. The remaining columns list the number of variables and constraints for the four models after preprocessing. The preprocessing eliminates for instance dominated constraints and dominated and infeasible frequency assignments. For example a frequency $f$ is dominated for line $l$ if $f > \max_{e \in l}\{F(e)\}$.

The instances were solved using the constraint integer programming framework SCIP version 3.0.1 [17] with CPLEX 12.5 as LP-solver. We set a time limit of 1 hour for all instances and used the default settings of SCIP, apart from the primal heuristic "shiftandpropagate" which we turned off. All computations were done on an Intel(R) Xeon(R) CPU E3-1290, 3.7 GHz computer (in 64 bit mode) with 13 MB cache, running Linux and 16 GB of memory. The results are shown in Table 2.

The computations show that the set cover, symmetric band, and MIR cuts indeed improve the standard model. The superiority of model (QLP) does not always show up, because its root LP cannot be solved within one hour for those instances where the number of configuration variables is more than 10 times higher than the number of line and frequency variables. For all other instances, the dual bounds after solving the root node for (SLP$^{\mathrm{Q}}$) and (QLP) are better than those for (SLP$^{+}$). Model (SLP$^{\mathrm{Q}}$) is performing best on nearly all instances. Except for Chicago it has a better dual bound after terminating the computations than models (SLP) and (SLP$^{+}$). Hence, model (SLP$^{\mathrm{Q}}$) is a good compromise between improving the formulation with configuration variables and keeping the size of the formulation small.

**Acknowledgment** We thank three referees for helpful comments and suggestions.

## References

1 Ralf Borndörfer, Isabel Friedow, and Marika Karbstein. Optimierung des Linienplans 2010 in Potsdam. *Der Nahverkehr*, 30(4):34–39, 2012.

2 Ralf Borndörfer, Martin Grötschel, and Marc E. Pfetsch. A column-generation approach to line planning in public transport. *Transportation Science*, 41(1):123–132, 2007.

3 Ralf Borndörfer and Marika Karbstein. A direct connection approach to integrated line planning and passenger routing. In Daniel Delling and Leo Liberti, editors, *ATMOS 2012 - 12th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and*

*Systems*, OpenAccess Series in Informatics (OASIcs), pages 47–57, Dagstuhl, Germany, 2012. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

**4** Ralf Borndörfer, Markus Reuther, Thomas Schlechte, and Steffen Weider. A Hypergraph Model for Railway Vehicle Rotation Planning. In Alberto Caprara and Spyros Kontogiannis, editors, *11th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2011)*, volume 20 of *OpenAccess Series in Informatics (OASIcs)*, pages 146–155, Dagstuhl, Germany, 2011. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ZIB Report 11-36.

**5** Ralf Borndörfer and Thomas Schlechte. Models for railway track allocation. In Christian Liebchen, Ravindra K. Ahuja, and Juan A. Mesa, editors, *ATMOS 2007 – 7th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems*, Dagstuhl, Germany, 2007. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany. ZIB Report 07-02.

**6** Michael Bussieck. Gams – lop.gms: Line optimization. `http://www.gams.com/modlib/libhtml/lop.htm`.

**7** Michael Bussieck. *Optimal Lines in Public Rail Transport.* PhD thesis, Teschnische Universität Braunschweig, 1998.

**8** Michael R. Bussieck, Peter Kreuzer, and Uwe T. Zimmermann. Optimal lines for railway systems. *Eur. J. Oper. Res.*, 96(1):54–63, 1997.

**9** M. T. Claessens, Nico M. van Dijk, and Peter J. Zwaneveld. Cost optimal allocation of rail passanger lines. *European Journal of Operations Research*, 110(3):474–489, 1998.

**10** Sanjeeb Dash, Oktay Günlük, and Andrea Lodi. MIR closures of polyhedral sets. *Mathematical Programming*, 121(1):33–60, 2010.

**11** Jan-Willem Goossens, Stan van Hoesel, and Leo Kroon. On solving multi-type line planning problems. METEOR Research Memorandum RM/02/009, University of Maastricht, 2002.

**12** Jan-Willem Goossens, Stan van Hoesel, and Leo Kroon. A branch-and-cut approach for solving railway line-planning problems. *Transportation Science*, 28(3):379–393, 2004.

**13** Marika Karbstein. *Line Planning and Connectivity.* PhD thesis, TU Berlin, 2013.

**14** Christian Raack. *Capacitated Network Design – Multi-Commodity Flow Formulations, Cutting Planes, and Demand Uncertainty.* Phd thesis, TU Berlin, 2012.

**15** Anita Schöbel. Line planning in public transportation: models and methods. *OR Spectrum*, pages 1–20, 2011.

**16** Anita Schöbel and Susanne Scholl. Line planning with minimal traveling time. In Leo G. Kroon and Rolf H. Möhring, editors, *Proceedings of 5th Workshop on Algorithmic Methods and Models for Optimization of Railways*, 2006.

**17** SCIP – Solving Constraint Integer Programs. `http://scip.zib.de`.

**18** Stadtwerke Potsdam – ViP Verkehrsbetrieb Potsdam GmbH. Vip website. `http://vip-potsdam.de`.

**19** Mechthild Stoer and Geir Dahl. A polyhedral approach to multicommodity survivable network design, 1994.

**20** Transportation network test problems. `http://www.bgu.ac.il/~bargera/tntp/`.

**21** Laurence A. Wolsey. Valid inequalities for 0-1 knapsacks and MIPs with generalized upper bound constraints. *Discrete Applied Mathematics*, 29:251–261, 1990.

**22** Laurence A. Wolsey. *Integer Programming.* John Wiley & Sons, first edition, 1998.

# The Stop Location Problem with Realistic Traveling Time *

Emilio Carrizosa[1], Jonas Harbering[2], and Anita Schöbel[2]

1 Universidad de Sevilla
C/ Tarfia s/n, 41012 Sevilla, Spain
ecarrizosa@us.es

2 Georg-August Universität Göttingen
Lotzestraße 16–18, 37083 Göttingen, Germany
{jo.harbering,schoebel}@math.uni-goettingen.de

**Abstract**

In this paper we consider the location of stops along the edges of an already existing public transportation network. This can be the introduction of bus stops along some given bus routes, or of railway stations along the tracks in a railway network. The positive effect of new stops is given by the better access of the customers to the public transport network, while the traveling time increases due to the additional stopping activities of the trains which is a negative effect for the customers.

Our goal is to locate new stops minimizing a *realistic* traveling time which takes acceleration and deceleration of the vehicles into account. We distinguish two variants: in the first (academic) version we locate $p$ stops, in the second (real-world applicable) version the goal is to *cover* all demand points with a minimal amount of realistic traveling time. As in other works on stop location, covering may be defined with respect to an arbitrary norm. For the first version, we present a polynomial approach while the latter version is NP-hard. We derive a finite candidate set and an IP formulation. We discuss the differences to the model neglecting the realistic traveling time and provide a case study showing that our procedures are applicable in practice and do save in average more than 3% of traveling time for the passengers.

**1998 ACM Subject Classification** G.1.6 Optimization

**Keywords and phrases** Stop Location, Realistic Traveling Time, IP Formulation

**Digital Object Identifier** 10.4230/OASIcs.ATMOS.2013.80

## 1 Introduction

The acceptance of public transportation depends on various components such as convenience, punctuality, reliability, etc. In this paper, we address the question of convenience for the passengers. In particular, we investigate the problem of establishing additional stops (or stations) which on the one hand guarantee a good accessibility to the transportation network, but on the other hand do not increase the traveling time of passengers too much.

Due to their great potential for improving public transportation systems, several versions of the *stop location problem* (also called *station location problem*) have been considered by various authors in the last years, see [16] for a survey. In order to find "good" locations for

* The first author is partially supported by MTM2012-36163-C06-03 and P11-FQM-7603. The other authors are partially supported by the European Union Seventh Framework Programme (FP7-PEOPLE-2009-IRSES) under grant number 246647 and the New Zealand Government (project OptALI).

new stops, several objective functions are possible. One of the most frequently discussed goals is to minimize the number of stops such that each demand point is within a tolerable distance from at least one stop. The maximal distance that a customer is willing to tolerate is called *covering radius*, hence we call this type of stop location problem *SL-Cov* for short. For bus stops a covering radius of 400 m is common. In rail transportation, the covering radius is much larger (at least 2 km).

In the literature, stop location problems have been introduced in [1] and considered in [12, 10, 11, 6, 18], see also references therein. In these papers, the problem is treated in a discrete setting, i.e., a finite set is considered as potential new stops. [17] allow a continuous set of possible locations for the stops, for instance, all points on the current bus routes or railway tracks. An application of this *continuous* version is given in [2], where the authors report on a project with the largest German rail company (Deutsche Bahn) and consider the trade-off between the positive and negative effects of stops. The negative effect of longer traveling times due to additional stops is compared with the positive effect of shorter access times, the goal is to maximize the difference of the two effects.

Based on this application, variants of the continuous stop location problem have been treated in [5, 16, 17]. The problem has been solved for the case of two intersecting lines, see [7]. Algorithmic approaches for solving the underlying covering problem have been studied in [15, 9]. Complexity and approximation issues have been presented in [8].

Another objective function is to minimize the sum of distances from the customers to the public transportation system, i.e. the sum of the distances between the demand facilities and their closest stops, see [13, 14]. Recently, covering a set of OD-pairs with a given number of stops has been studied, see e.g., [4] and references therein.

*Contribution.* All the mentioned papers use a rough approximation of the traveling time by adding a penalty for each stop. Since trains have a long acceleration and deceleration phase this is unrealistic in practice. In this paper we consider stop location problems with *realistic* traveling time.

*Structure of the paper.* We develop and analyze the realistic traveling time function in Section 2. We then consider two variants of the stop location problem with realistic traveling time. In Section 3 we want to locate $p$ stops minimizing the traveling time, while in Section 4 we want to cover all demand points with a set of stops, again with minimal realistic traveling time. While we present a polynomial algorithm for the former problem, the latter problem is NP hard. Nevertheless we are able to develop a finite dominating set which is the basis for an integer programming formulation. We compare our new model to the existing covering models (without realistic traveling time) and present a case study with numerical results. All proofs can be found in the appendix.

## 2 Stop location with realistic traveling time

In the stop location problems considered so far, the traveling time for passengers due to new stops is estimated by adding a penalty $\text{time}_{\text{pen}}$ for every stop to be located. This is an exact estimate if the distance between two stops is larger than the distance needed for acceleration and deceleration, and if $\text{time}_{\text{pen}}$ gives the loss of traveling time resulting from the additional stop. As an example, $\text{time}_{\text{pen}}$ is estimated as two minutes for German regional trains. However, since trains accelerate slowly, this estimate is not realistic if the distance between two stops is rather short.

In this section we hence first introduce a function describing the realistic traveling time of a train between two consecutive stops. This function depends on the distance $d$ between those two consecutive stops. Being able to compute realistic traveling times, we then define two variants of the stop location problem, both with realistic traveling time.

▶ **Lemma 1.** *[see also [3]] Let a maximum cruising speed $v_0 > 0$, an acceleration of $a_0 > 0$ and a deceleration of $b_0 > 0$ of a vehicle be given. Then the traveling time function depending on $d$, where $d$ is the distance between two consecutive stops, is given as*

$$T(d) = \begin{cases} \sqrt{\frac{2(a_0+b_0)}{a_0 b_0} d} & \text{if } d \leq d^{max}_{v_0,a_0,b_0} \\ \frac{d}{v_0} + \frac{v_0}{2a_0} + \frac{v_0}{2b_0} & \text{if } d \geq d^{max}_{v_0,a_0,b_0} \end{cases} \quad \text{where } d^{max}_{v_0,a_0,b_0} = \frac{v_0^2}{2a_0} + \frac{v_0^2}{2b_0}$$

The formula is a simple consequence from Newton's laws of motion. E.g., in [3] the traveling time function for the case $a_0 = b_0$ is introduced, and the practical relevance of this better estimate is analyzed for fire engines in New York City.

Note that $d^{max}_{v_0,a_0,b_0}$ is the point where the traveling time function turns from a square root behavior to a linear behavior.

The shape and exact values of the function can be easily calculated; its main properties can be verified straightforwardly.

▶ **Lemma 2.** *$T(d)$ is continuous, differentiable, concave and monotonically increasing. Furthermore, for any $d$ we have $\sqrt{\frac{2(a_0+b_0)}{a_0 b_0} d} \leq \frac{d}{v_0} + \frac{v_0}{2a_0} + \frac{v_0}{2b_0}$.*

The properties of $T$ can be shown by easy calculations.

In the two variants of the stop location problem our objective is to minimize the (realistic) traveling time which is determined as follows.

Let $G = (V, E)$ be the given network in which the new stops should be located. Let $e = (i, j) \in E$ be an edge with length $d_e$. A point $s = (e, x) \in e$ is defined as the point on edge $e$ with distance $d(i, s) = x$ and distance $d(s, j) = d_e - x$, $0 \leq x \leq d_e$. Note that $i = (e, 0)$ and $j = (e, d_e)$. The set of points of $G$ is denoted as $\mathcal{S} = \bigcup_{e \in E} e$. The set of points between two points $s_1 = (e, x_1)$ and $s_2 = (e, x_2)$ on the same edge is denoted as $[s_1, s_2] = \{(e, x) : x_1 \leq x \leq x_2\}$.

A new stop $s$ in the network may be any point $s = (e, x)$. We assume that all vertices $V$ are existing stops.

Given a set $S \in \mathcal{S}$ of points of $G$, every set $S_e = S \cap e = \{s_1, \ldots, s_p\} \subseteq e$ of points on $e = (i, j)$ can be naturally ordered along the edge $e$ such that $d(s_1, i) \leq \ldots \leq d(s_p, i)$. Let $\leq_e$ denote this ordering. Adding the points of $S$ as new stops gives a subdivision of the network $G$, i.e. a new network $(V \cup S, E(S))$ (see Figure 2), where

$$E(S) = \{(s_i, s_j) : s_i = (e, x_i) \text{ and } s_j = (e, x_j) \text{ are consecutive on some } e \in E \text{ w.r.t. } \leq_e\}.$$

The length of an edge $e' = ((e, x_i), (e, x_j)) \in E(S)$ is given as $d_{e'} = |x_j - x_i|$.

Finally, given a set $S$ of points on the graph $G$, we can define the (realistic) traveling time function as

$$g(S) := \sum_{e' \in E(S)} T(d_{e'}).$$

The stop location problem (SL) on $G = (V, E)$ is to locate a set of stops $S$ which are points on $G$. Our objective is to minimize the (realistic) traveling time $g(S)$. This function can be seen as an intrinsical property of the network and an estimation for the traveling time of passengers without having information of their real paths and demands.

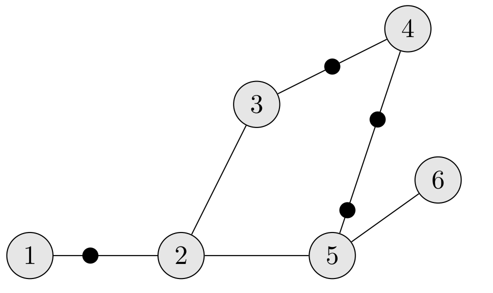


**Figure 1** Locating $p = 4$ new stops on a Network $G = (V, E)$.

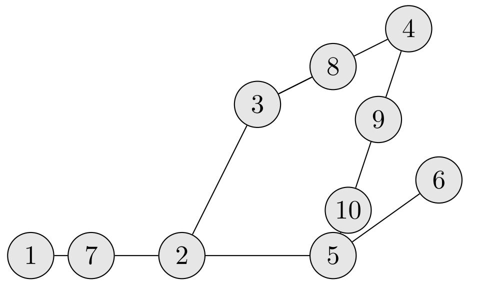


**Figure 2** The new network $(V \cup S, E(S))$ with $p = 4$ new stops.

Without any constraints, $S = \emptyset$ would be the trivial optimal solution. We hence need to ensure that enough new stops are located. We consider the following two possibilities:

**(SL-TT-p)** Here, the goal is to locate $p$ new stops on $G$. It is further required that the minimal distance between two stops is at least $\epsilon$, i.e., $d_{e'} \geq \epsilon$ for all $e' \in E(S)$.

**(SL-TT-Cov)** This is an extension of the stop location problems considered in the literature, in which it is assumed that the network is embedded in the plane $\mathbb{R}^2$, and that a finite set of demand points $\mathcal{P} \subseteq \mathbb{R}^2$ is given. Furthermore, to measure the access times from the demand points to the railway network, a distance function $dist : \mathbb{R}^2 \times \mathbb{R}^2 \to \mathbb{R}$ is given which has been derived from a norm, i.e. $dist(x, y) = \|y - x\|$ for a given norm $\| \cdot \|$. For a set $S \in \mathcal{S}$ we can now define the set of covered demand points as

▶ **Definition 3.** $cover_{\mathcal{P}}(S) = \{p \in \mathcal{P} : dist(p, s) \leq r \text{ for some } s \in \mathcal{S}\}$.

In (SL-TT-Cov) we look for a set $S$ covering all demand points (i.e. $cover_{\mathcal{P}}(S) = \mathcal{P}$).

## 3 (SL-TT-p) Locating p stops

We start with locating a fixed number of $p$ stops on a single line segment. We hence have $G = (V, E)$ where $V = \{i, j\}$ is the set of nodes and $E = \{e = (i, j)\}$ is one edge. Locating $p$ new stops $S$ on the edge $e$ increases the traveling time for the passengers that want to traveling from $i$ to $j$. Our goal is to minimize this traveling time using the realistic traveling time function $g(S)$:

**(Line-SL-TT-p)** Let $G = (\{i, j\}, \{e\})$ be one single edge, $v_0 > 0$, $a_0 > 0$, and $b_0 > 0$ and let a natural number $p > 0$, and $0 \leq \epsilon \leq \frac{d_e}{p+1}$ be given. Find a subset $S^* \subseteq \mathcal{S}$ with $|S^*| = p$ and $d_{e'} \geq \epsilon$ for all $e \in E(S^*)$ such that $g(S^*)$ is minimized.

Note that (Line-SL-TT-p) is not feasible if $d_e < (p + 1)\epsilon$.

We start by discussing the case of locating only $p = 1$ stop without further restriction (i.e. $\epsilon = 0$) since this instance contains the main idea for the general case.

Let $d_e$ be the length of edge $e = (i, j)$. Let $s = (e, x)$ be a new stop on $e$. We want to determine $x$. Given any $x \in [0, d_e]$, the traveling time for passengers from $i$ to $j$ is

$$g(S) = g(x) = T(x) + T(d_e - x),$$

i.e., we have to solve $\min\{T(x) + T(d_e - x) : 0 \leq x \leq d\}$. Since $T$ is a concave function, also $T(d_e - x)$ is concave, hence also $g(x)$. A concave function over a compact interval takes its minimum at one of the endpoints of the interval, we hence evaluate $g(0) = g(d_e) = T(0)+T(d_e)$ and obtain

▶ **Lemma 4.** *The only optima for (Line-SL-TT-p) for the case of locating $p = 1$ stop without a minimal distance (i.e. $\epsilon = 0$) are obtained at $s_1 = i$ and at $s_2 = j$.*

However, since $i$ and $j$ are already stops, this solution does not give any new stop and consequently is not what we want. We hence require a minimal distance of $\epsilon > 0$ between any pair of stops. This means to solve $\min\{T(x) + T(d - x) : \epsilon \le x \le d - \epsilon\}$ and again results in two optima on the boundary of the interval. We obtain:

▶ **Lemma 5.** *The only optima for (Line-SL-TT-p) for the case of locating $p = 1$ stop on the edge $e = (i, j)$ are obtained at $s_1 = (e, \epsilon)$ and $s_2 = (e, d_e - \epsilon)$.*

This result can be generalized for the case of locating $p \ge 2$ stops $s_1 = (e, x_1), \dots, s_p = (e, x_p)$. We hence look for the values of $x_1, \dots, x_p$.

Fixing $x_0 = 0$ and $x_{p+1} = d_e$, the respective optimization program is stated as

$$\begin{aligned} \min & \sum_{l=1}^{p+1} T(x_l - x_{l-1}) \\ \text{s.t. } & x_l - x_{l-1} \ge \epsilon \quad \forall\, l = 1, \dots, p+1 \\ & x_l \in \mathbb{R} \quad \forall l = 1 \dots, p \end{aligned}$$

First note, that for $\epsilon \ge d^{max}_{v_0,a_0,b_0}$ there is not much to worry about.

▶ **Lemma 6.** *If $\epsilon \ge d^{max}_{v_0,a_0,b_0}$, every feasible solution to (Line-SL-TT-p) has the same objective value.*

For $\epsilon < d^{max}_{v_0,a_0,b_0}$ we then find the following result.

▶ **Lemma 7.**
- *If $\frac{d_e}{p+1} < \epsilon < d^{max}_{v_0,a_0,b_0}$, any solution where all but two stops are at $\epsilon$-distance to both of their neighbors, and the remaining two are at $\epsilon$-distance to one of their neighbors, is optimal.*
- *If $\frac{d_e}{p+1} = \epsilon < d^{max}_{v_0,a_0,b_0}$, the unique solution where all stops are at $\epsilon$-distance to both of their neighbors is optimal.*

We summarize that the $p$ stops to be located are clustered together in an optimal solution along one edge. As can be seen easily, this behavior still holds if a complete network is given and $p$ stops should be located there. Obviously such a solution is not realistic for practical purposes. Thus, in the following a different model will be considered which is more related to realistic needs.

## 4 (SL-TT-Cov) Covering all demand points

### 4.1 Feasibility and complexity of (SL-TT-Cov)

As seen in the previous section it does not make much sense just to add $p$ new stops to an existing network. The main objective function used for locating new stops is usually a covering-type objective: With the new stops, one tries to cover as much demand as possible. Given a set of demand points $\mathcal{P}$ in the plane, we say that $p \in \mathcal{P}$ is covered by a set of stops $S \in \mathcal{S}$ if $dist(p, s) \le r$ for some $s \in S$, where $r$ is a fixed *covering radius*. The 'classic' stop location problem (SL-Cov) asks for a set of stops of minimal cardinality covering all demand points:

**(SL-Cov)** Let $G = (V, E)$ be a graph and a finite set of points $\mathcal{P} \subseteq \mathbb{R}^2$ be given. Find a subset $S^* \in \mathcal{S}$, such that $cover_{\mathcal{P}}(S^*) = \mathcal{P}$ and $|S^*|$ is minimized.

The goal of this section is is to cover all demand points with a set of stops $S$ such that the realistic traveling time function $g(S)$ is minimal:

**(SL-TT-Cov)** Let $G = (V, E)$ be a graph, $\mathcal{P} \subseteq \mathbb{R}^2$ be a finite set of points, $v_0 > 0$, $a_0 > 0$ and $b_0 > 0$ be given. Find a subset $S^* \in \mathcal{S}$, such that $cover_{\mathcal{P}}(S^*) = \mathcal{P}$ and $g(S^*)$ is minimized.

(SL-TT-Cov) need not be feasible, but if it is it admits a finite solution whose objective value can be bounded.

▶ **Lemma 8.**
- *(SL-TT-Cov) has a solution if and only if $cover_{\mathcal{P}}(\mathcal{S}) = \mathcal{P}$.*
- *If (SL-TT-Cov) has a feasible solution, then it also has a finite solution and $g(S^*) \leq (|E| + |\mathcal{P}|) \cdot \left(\max_{e \in E} \frac{d_e}{v_0} + \frac{v_0}{2a_0} + \frac{v_0}{2b_0}\right)$.*

While feasibility is easy to check, solving (SL-TT-Cov) is NP-hard.

▶ **Lemma 9.** *(SL-TT-Cov) is NP-hard.*

## 4.2 A finite dominating set for (SL-TT-Cov)

In the following we show that (SL-TT-Cov) can be reduced to a discrete problem by identifying a finite dominating set, i.e., a finite set of candidates $\mathcal{S}_{cand} \subseteq \mathcal{S}$, for which we know that it contains an optimal solution $S^*$, if the problem is feasible at all. Such a finite dominating set will enable us to derive an IP formulation in Section 4.3. It turns out that we can use nearly the same finite dominating set which has been used as candidate set for solving (SL-Cov) (see [17]). Throughout this section, let us assume that (SL-TT-Cov) is feasible, which can be tested (due to Lemma 8).

For an edge $e = (i, j) \in E$ we define

$$T^e(p) = \{s \in e : dist(p, s) \leq r\}$$

as the set of all points on the edge $e \subseteq \mathcal{S}$ that can be used to cover demand point $p$.

Since $T^e(p) = e \cap \{x \in \mathbb{R}^2 : dist(p, x) \leq r\}$ is the intersection of two convex sets, and contained in $e$, it turns out to be a line segment itself. This observation is due to [17].

▶ **Lemma 10** ([17]). *For each demand point $p \in \mathbb{R}^2$ the set $T^e(p)$ is either empty or an interval contained in edge $e$.*

Let $f_p^e, l_p^e$ denote the endpoints of the interval $T^e(p)$ (which may coincide with the endpoints $i, j$ of the edge $e$). We write $[f_p^e, l_p^e] = T^e(p)$. For each edge $e = (i, j)$ we define

$$\mathcal{S}_{cand}^e := \bigcup_{p \in \mathcal{P}} \{f_p^e, l_p^e\},$$

which can be ordered along the edge $e$ with respect to $\leq_e$. Let the resulting set be given as $\mathcal{S}_{cand}^e = \{s_0, s_1, \ldots, s_{N_e}\}$. In the following we show that

$$\mathcal{S}_{cand} = \bigcup_{e \in E} \mathcal{S}_{cand}^e$$

is a finite dominating set for (SL-TT-Cov).

From [17] we know that moving a point $s \in \mathcal{S}$ until it reaches an element of $\mathcal{S}_{cand}$ does not change $cover_{\mathcal{P}}(\{s\})$.

▶ **Lemma 11** ([17]). *Let $s \in e$ for an edge $e$ of $E$, and let $s_j, s_{j+1} \in \mathcal{S}_{cand}$ be two consecutive elements of the finite dominating set with $s_j <_e s <_e s_{j+1}$. Then*

$$cover_{\mathcal{P}}(\{s\}) \subseteq cover_{\mathcal{P}}(\{s_j\}) \cap cover_{\mathcal{P}}(\{s_{j+1}\}),$$

*in particular, the cover of $s$ does not decrease when moving $s$ between $s_j$ and $s_{j+1}$.*

Now we are able to prove that $\mathcal{S}_{cand} = \bigcup_{e\in E} \mathcal{S}^e_{cand}$ is, indeed, a finite dominating set.

▶ **Theorem 12.** *Either (SL-TT-Cov) is infeasible, or there exists an optimal solution $S^* \subseteq \mathcal{S}_{cand}$.*

The number of candidates $|\mathcal{S}_{cand}| \leq 2|E||\mathcal{P}|$. Thus iterating leads to a number of $\mathcal{O}\left(2^{2|E||\mathcal{P}|}\right)$ different solutions to be tested. In the following, this is done by integer programming.

## 4.3 An integer programming formulation for (SL-TT-Cov)

Let $A^{cov} = (a_{p,s})_{p\in\mathcal{P}, s\in\mathcal{S}_{cand}}$ denote the covering matrix, given by

$$a_{ps} = \begin{cases} 1 & \text{if } p \in cover_{\mathcal{P}}(\{s\}) \\ 0 & \text{otherwise.} \end{cases}$$

Furthermore, let $E_{cand} = \{(s, s') : s, s' \in \mathcal{S}_{cand} \cup V \text{ and } s, s' \in e \text{ for some edge } e \in E\}$ be the set of all possible edges obtained by building any set of stops $S \subseteq \mathcal{S}_{cand}$. For those edges the distance $d_{e'}, e' \in E_{cand}$ can be precalculated. The IP formulation of the discrete version of (SL-TT-Cov) is then given by

$$\min \quad \sum_{e\in E_{cand}} T(d_e) y_e \tag{1}$$

$$s.t. \sum_{s\in\mathcal{S}_{cand}} a_{ps} x_s \geq 1 \qquad \forall p \in \mathcal{P} \tag{2}$$

$$x_{s_i} + x_{s_j} - \sum_{\substack{s\in[s_i,s_j]\cap\mathcal{S}_{cand} \\ s\notin\{s_i,s_j\}}} x_s \leq y_{e'} + 1 \qquad \forall e' = (s_i, s_j) \in E_{cand} \tag{3}$$

$$x \in \{0,1\}^{|\mathcal{S}_{cand}|} \tag{4}$$

$$y \in \{0,1\}^{|E_{cand}|} \tag{5}$$

The variables $x_i$ and $y_e$ have the following interpretation.

$$x_{s_i} = \begin{cases} 1 & \text{if stop } s_i \in \mathcal{S}_{cand} \text{ is built.} \\ 0 & \text{otherwise.} \end{cases} \qquad y_e = \begin{cases} 1 & \text{if edge } e \in E_{cand} \text{ is built.} \\ 0 & \text{otherwise.} \end{cases}$$

Constraint (2) ensures that every demand point is covered by at least one stop. Constraints of type (3) ensure that an edge is considered in the objective function if and only if it is built, i.e. if and only if its two endpoints are stops and no candidate between the two endpoints is also a stop. Finally, the objective function (1) then gives the realistic traveling time:

▶ **Lemma 13.** *The above stated IP formulation is correct for (SL-TT-Cov).*

Since the proof is straight forward it is spared.

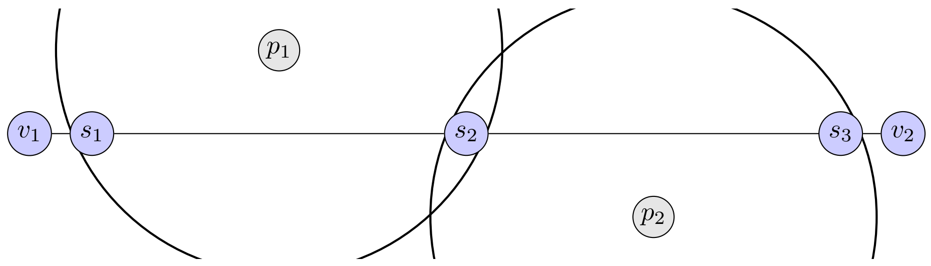


**Figure 3** Example for SL-TT-Cov with more stops built than for SL-Cov.

▶ Remark. Consider the case, where $d_e \geq d^{max}_{v_0,a_0,b_0}$ for all $e \in E_{cand}$. Then the objective function can be rewritten as

$$\sum_{e' \in E_{cand}} T(d_{e'})y_e = \sum_{e \in E} T(d_e) + \left(\frac{v_0}{2a_0} + \frac{v_0}{2b_0}\right) \sum_{s \in S_{cand}} x_s,$$

thus the variables $y_e$ can be eliminated, i.e., the objective function is equivalent to minimizing the number of new stops in this case. We conclude that (SL-Cov) and (SL-Cov-TT) are equivalent if $d_e \geq d^{max}_{v_0,a_0,b_0}$ for all $e \in E_{cand}$.

The number of constraints given by the candidate edges is of order $O(|\mathcal{S}_{cand}|^2)$:

▶ **Lemma 14.** *Suppose a network $G$=($V$,$E$) and $\mathcal{S}_{cand}$ are given. Let $|E| = m$ and $|\mathcal{S}_{cand}| = n = \sum_{i=1}^{m} n_i$, where $n_i$ for $i = 1, \dots, m$ is the number of candidate stops on edge $e_i$. Then the number of candidate edges is given by $|E_{cand}| = \sum_{i=1}^{m} \binom{n_i+2}{2}$*

Note that if there exists an $1 \leq i \leq m$ such that $n_i = 0$ then $T(d_{e_i})$ is a constant and thus does not have to be considered. In fact, the number of variables and constraints can then be reduced.

We close this section by a comparison between (SL-Cov) and (SL-Cov-TT). The following example shows a situation in which the realistic traveling time can be reduced by building two stops instead of only one.

▶ **Example 15.** In Figure 3 two demand points $p_1$ and $p_2$ have to be covered by stops on $e = (v_1, v_2)$. In order to minimize the number of stops it is sufficient to build only one stop, namely $s_2$, i.e., $s = \{s_2\}$ is an optimal solution. We compare $S$ with the solution $\tilde{S} = \{s_1, s_2\}$, where $s_1$ and $s_3$ are close enough to $v_1$ and $v_2$ respectively. Assuming $d_{v_1,s_2}, d_{s_2,v_2} \geq d^{max}_{v_0,a_0,b_0}$, the traveling times can be computed as

$$\begin{aligned}
f(S) &= T(d_{v_1,s_2}) + T(d_{s_2,v_2}) = \frac{d_e}{v_0} + \frac{v_0}{a_0} + \frac{v_0}{b_0} \\
f(\tilde{S}) &= T(d_{v_1,s_1}) + T(d_{s_1,s_3}) + T(d_{s_3,v_2}) \\
&= \sqrt{\frac{2(a_0+b_0)}{a_0 b_0} d_{v_1,s_1}} + \frac{d_e - d_{v_1,s_1} - d_{s_3,v_2}}{v_0} + \frac{v_0}{2a_0} + \frac{v_0}{2b_0} + \sqrt{\frac{2(a_0+b_0)}{a_0 b_0} d_{s_3,v_2}}
\end{aligned}$$

and by letting $d_{v_1,s_1}$ and $d_{s_3,v_2}$ tend to 0, we see that $f(\tilde{S}) < f(S)$.

Other examples of the same pattern can be constructed which show that the number of stops in an optimal solution to (SL-TT-Cov) can differ by more than one from the number of stops in an optimal solution to (SL-Cov).

## 5 Experiments

*Environment.* All experiments were conducted on a PC with 24 six-core Intel Xenon X5650 Processor running at 2.67 GHz with 12 MB cache and a main memory of 94 GB. IPs were solved using Xpress Optimizer v23.01.05. The running time limit of the solver was set to 300 seconds.

*Benchmark set.* The southern part of the existing railway network of Lower Saxony, Germany, is used as the existing network $G = (V, E)$. From the same area the 34 largest cities are considered as demand points if they are not already close enough to an existing stop. This is the setting for the first benchmark set (LS=Lower Saxony). For our second benchmark set, stops which have only two adjacent tracks are removed and thus a set (LSR=Lower Saxony Reduced) with longer tracks and more uncovered demand points is obtained. This set has higher complexity.

The values for the traveling time are chosen according to the real properties, which are acceleration and deceleration of $0.7m/s^2$ and a cruising speed of $200km/h$. For a set of different radii ($r \in \{1750, 2100, 2450, \dots, 12950\}$ (in meters)), we constructed instances containing all demand points which can be covered by $r$, i.e. $\mathcal{P}$ increases with the radius.

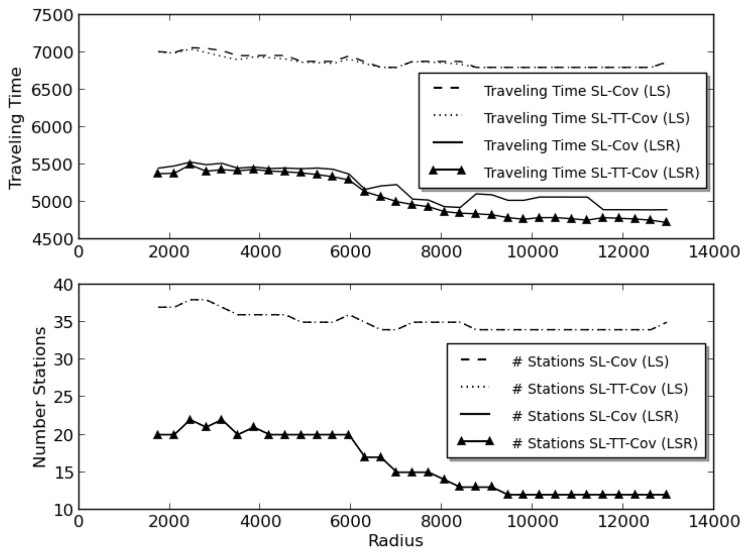


**Figure 4** Traveling time with respect to the number of built stops.

*Setup.* The quality of the IP formulations for (SL-TT-Cov) and for (SL-Cov) (see [17] for the IP formulation of (SL-Cov)) are compared. To this end, for each of the benchmark sets and every radius $r \in \{1750, 2100, 2450, \dots, 12950\}$ both models have been solved by Xpress Optimizer. Then for each run the quality of the solution is measured by evaluating the traveling time and the number of stops built.

*Hypotheses.* The evaluations are designed to approve or disprove the below stated hypotheses.

1. (SL-TT-Cov) performs better than (SL-Cov) in terms of traveling time.
2. (SL-TT-Cov) performs worse than (SL-Cov) in terms of number of stops.

**Table 1** Average values of the objective functions for the solutions of (SL-Cov) and (SL-TT-Cov).

| Set (LS) | SL-Cov | SL-TT-Cov | Set (LSR) | SL-Cov | SL-TT-Cov |
|---|---|---|---|---|---|
| Traveling Time $g(S)$ | 6878.62 | 6863.34 | Traveling Time $g(S)$ | 5201.62 | 5069.24 |
| # Built stops $\|S\|$ | 35.09 | 35.09 | # Built stops $\|S\|$ | 16.06 | 16.06 |

3. With increasing radius and a fixed set of demand points, the traveling time decreases.
4. The difference in performance between (SL-TT-Cov) and (SL-Cov) is more evident on (LSR) than on (LS).
5. The running time of (SL-TT-Cov) is exponential in the number of candidates.
6. As the acceleration tends to infinity, the traveling time of the solutions of (SL-TT-Cov) and (SL-Cov) tend to the traveling time with constant speed.

*Results.* Table 1 summarizes our results calculating the average values of the two objective functions for all instances.

Hypotheses 1 and 2. Considering the benchmark sets (LS) and (LSR) the solutions of (SL-Cov) and (SL-TT-Cov) in terms of the number of built stops do not vary at all. In terms of the resulting traveling time on (LS) only small differences are recognizable. However, on the benchmark set (LSR) the results show bigger differences. The average traveling time of (SL-Cov) can be reduced by more than 3% by using (SL-TT-Cov). These messages can clearly be confirmed by Figure 4. Hence, from the experiments we can approve hypotheses 1 and disprove 2.

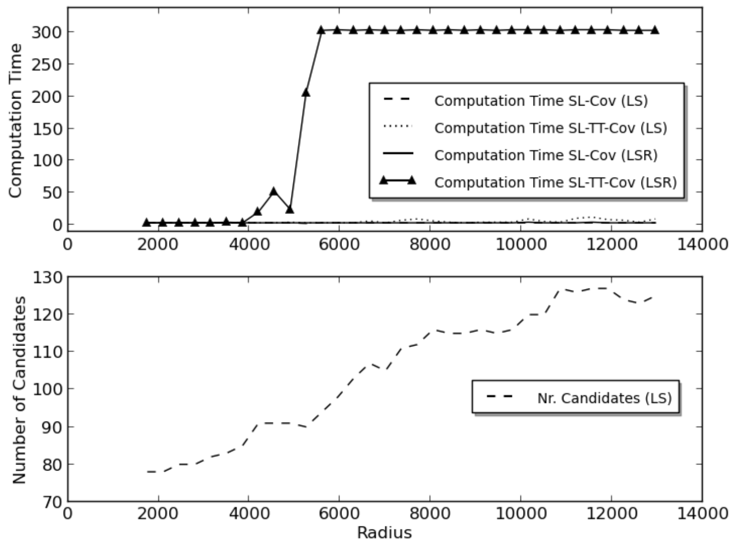


**Figure 5** Computing time with respect to the number of candidates.

Hypothesis 3. Figure 4 clearly shows that in both sets (LS) and (LSR) the traveling time decreases with increasing radius $r$. This makes sense as for the same demand points but increasing radius possibly more demand points can be covered by the same stop. In some instances the number of stops or the traveling time increases with the radius, which is due to the increased number of demand points which require more stops to be built. In general though, we can detect that the traveling time decreases with increasing radius. Thus, the hypothesis 3 can be approved.

Hypothesis 4. Also hypothesis 4 can be approved by the results depicted in Figure 4. Note that the vertices in the underlying network $G$ are always stops. Thus, for (LS) already 35 stops are fixed to be built, which explains why there is no big difference between (SL-Cov) and (SL-TT-Cov). For (LSR) though, the number of previously fixed stops is only 10, i.e., the models (SL-Cov) and (SL-TT-Cov) have more freedom to find a solution and hence the difference in terms of the traveling time is bigger. Hypothesis 4 can hence be approved.

Hypothesis 5. On the other hand the more freedom is granted to the models, the higher is the complexity and subsequently the higher is the running time. Figure 5 depicts the running time of (SL-Cov) and (SL-TT-Cov) for (LS) and (LSR). The maximal running time for (SL-TT-Cov) is set to 300 seconds and the solution obtained if the algorithm exceeds this limit is usually not optimal. In the experiments, all solution obtained were at least feasible, and although not necessarily optimal, the solutions for (SL-TT-Cov) have lower traveling times than the solutions for (SL-Cov). Figure 5 hence approves the hypothesis 5.

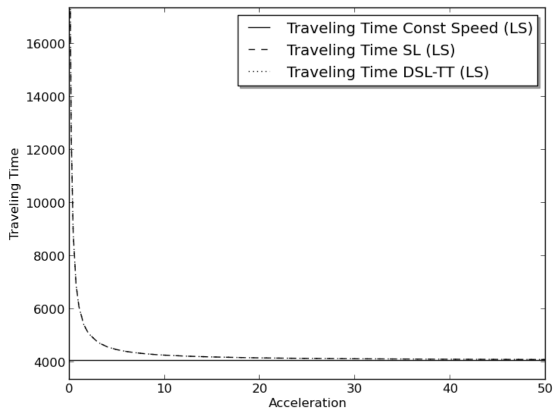


**Figure 6** Relation between traveling time and acceleration

Hypothesis 6. Finally, Figure 6 depicts the behavior of the traveling time for increasing acceleration and deceleration. To this end, we solved (SL-Cov) and (SL-TT-Cov) on (LS) with a fixed radius of 3500 meters for different acceleration and deceleration values. It is assumed that acceleration and deceleration are always equal. The traveling time is compared to the function summing up all edge lengths and dividing by the cruising speed. This is the traveling time function assuming a constant speed. For increasing acceleration we can detect that the traveling time $T$ tends to the value of the traveling time assuming constant speed. Taking into account the shape of $T(d)$ it means that for increasing $a_0$ and $b_0$ the acceleration and deceleration phases become shorter. Thus, hypothesis 6 can be approved.

## 6 Conclusion and further research

In this paper we included a realistic traveling time function in stop location problems. We derived a finite dominating set and an IP formulation and showed the applicability of the model on two different benchmark sets. It turns out that the solutions of (SL-TT-Cov) usually outperform the solutions of (SL-Cov) with a trade-off of having higher running times.

Further research on this topic goes into two directions. First, we assumed that all vertices of the existing network are built as stops. However, it may be better to close or move some of these. In order to model this appropriately, an integration with line planning is necessary. Secondly, the traveling time for the passengers could be even more realistic if OD-pairs are considered. Minimizing their traveling time leads to a different model and thus analysis.

## References

1 J. Gleason. A set covering approach to bus stop allocation. *Omega*, 3:605–608, 1975.

2 H.W. Hamacher, A. Liebers, A. Schöbel, D. Wagner, and F. Wagner. Locating new stops in a railway network. *Electronic Notes in Theoretical Computer Science*, 50(1), 2001.

3 P. Kolesar, W. Walker, and J. Hausner. Determining the relation between fire engine travel times and travel distances in new york city. *Operations Research*, 23:614–627, 1975.

4 M.-C. Körner, J.A. Mesa, F. Perea, A. Schöbel, and D. Scholz. A maximum trip covering location problem with an alternative mode of transportation on tree networks and segments. *TOP*, 2012. published online, DOI 10.1007/s11750-012-0251-y.

5 E. Kranakis, P. Penna, K. Schlude, D.S. Taylor, and P. Widmayer. Improving customer proximity to railway stations. In *CIAC*, pages 264–276, 2003.

6 G. Laporte, J.A. Mesa, and F.A. Ortega. Locating stations on rapid transit lines. *Computers and Operations Research*, 29:741–759, 2002.

7 M.F. Mammana, S. Mecke, and D. Wagner. The station location problem on two intersecting lines. *Electronic Notes in Theoretical Computer Science*, 92:52–64, 2004.

8 S. Mecke, A. Schöbel, and D. Wagner. Stop location - complexity and approximation issues. In *5th workshop on algorithmic methods and models for optimization of railways*, number 06901 in Dagstuhl Seminar proceedings, 2006.

9 S. Mecke and D. Wagner. Solving geometric covering problems by data reduction. In *Proceedings of European Symposium on Algorithms (ESA)*, pages 760–771, 2004.

10 A. Murray. Strategic analysis of public transport coverage. *Socio-Economic Planning Sciences*, 35:175–188, 2001.

11 A. Murray. A coverage models for improving public transit system accessibility and expanding access. *Annals of Operations Research*, 123:143–156, 2003.

12 A. Murray, R. Davis, R.J. Stimson, and L. Ferreira. Public transportation access. *Transportation Research D*, 3(5):319–328, 1998.

13 A. Murray and X. Wu. Accessibility tradeoffs in public transit planning. *J. Geographical Syst.*, 5:93–107, 2003.

14 D. Poetranto, H.W. Hamacher, S. Horn, and A. Schöbel. Stop location design in public transportation networks: Covering and accessibility objectives. *TOP*, 17(2):335–346, 2009.

15 N. Ruf and A. Schöbel. Set covering problems with almost consecutive ones property. *Discrete Optimization*, 1(2):215–228, 2004.

16 A. Schöbel. *Optimization in public transportation. Stop location, delay management and tariff planning from a customer-oriented point of view.* Optimization and Its Applications. Springer, New York, 2006.

17 A. Schöbel, H.W. Hamacher, A. Liebers, and D. Wagner. The continuous stop location problem in public transportation. *Asia-Pacific Journal of Operational Research*, 26(1):13–30, 2009.

18 C. Wu and A. Murray. Optimizing public transtit quality and system access: the multiple-route, maximal covering/shortest path problem. *Environment and Planning B: Planning and Design*, 32:163–178, 2005.

## A Appendix

**Proof.** (Lemma 6) Let $S = \{(e, x_1), \ldots, (e, x_p)\}$ with $x_1 < \ldots < x_p$ be a feasible solution, and let $x_0 = 0$ and $x_{p+1} = d_e$. Then

$$g(S) = \sum_{l=1}^{p+1} T(x_l - x_{l-1}) = \sum_{l=1}^{p+1} \frac{(x_l - x_{l-1})}{v_0} + \frac{v_0}{2a_0} + \frac{v_0}{2b_0} = \frac{d_e}{v_0} + \frac{v_0(p+1)}{2}\left(\frac{1}{a_0} + \frac{1}{b_0}\right),$$

which is independent of $S$. ◀

**Proof.** (Lemma 7) Note that $T(x - y)$ is a concave function in $(x, y)$ on $\{(x, y) : x \geq y\}$, hence $g(x_1, \ldots, x_p) = \sum_{l=1}^{p+1} T(x_l - x_{l-1})$ is also concave. The minimum of the above program is hence taken at an extreme point of the feasible polyhedral set $P = \{(x_1, \ldots, x_p) : x_l + \epsilon \leq x_{l+1}, l = 0, \ldots, p\}$. $P$ has exactly $p + 1$ extreme points given by

$$x^h = (x_0 + \epsilon, x_0 + 2\epsilon, \ldots, x_0 + (h-1)\epsilon, x_{p+1} - (p - (h-1))\,\epsilon, \ldots, x_{p+1} - 2\epsilon, x_{p+1} - \epsilon)$$

for $h = 1, \ldots, p + 1$. Evaluating the objective function at an extreme point $x^h$ yields

$$\begin{aligned} g(x^h) &= \sum_{i=0}^{p} T(x^h_{i+1} - x^h_i) \\ &= pT(\epsilon) + T\left(x_{p+1} - p\epsilon + (h-1)\epsilon - x_0 - (h-1)\epsilon\right) = pT(\epsilon) + T(d_e - p\epsilon) \end{aligned}$$

which is independent of $h$. Hence, any of the extreme points is optimal. ◀

**Proof.** (Lemma 8) The first part of the lemma is obvious. For the second part, let (SL-TT-Cov) be feasible. Then there exists some point $s_p \in \mathcal{S}$ such that $dist(p, s) \leq r$ for every demand point $p \in \mathcal{P}$. Choose $S := \{s_p : p \in \mathcal{P}\}$ as a feasible solution. Each stop $s \in S$ adds a new edge to $E(S)$, hence $|E(S)| = |E| + |\mathcal{P}|$. Let $e' = (i, j) \in E(S)$ be a new edge with $i = (\bar{e}, x_i), j = (\bar{e}, x_j)$ for some $\bar{e} \in E$. Then we estimate $d_{e'} \leq d_{\bar{e}} \leq \max_{e \in E} d_e$, and since $T$ is monotone we obtain

$$T(d_{e'}) \leq \max_{e \in E} T(d_e) \overset{(I)}{\leq} \max_{e \in E} \frac{d_e}{v_0} + \frac{v_0}{2a_0} + \frac{v_0}{2b_0}, \text{ where (I) is a result from Lemma 2.}$$

$$\text{Hence, } g(S^*) \leq g(S) \leq |E(S)| \max_{e \in E} T(d_e) \leq (|E| + |\mathcal{P}|) \max_{e \in E} \frac{d_e}{v_0} + \frac{v_0}{2a_0} + \frac{v_0}{2b_0}.$$

◀

**Proof.** (Lemma 9) To see that (SL-TT-Cov) is NP-hard, we reduce it to the discrete stop location in a network: Given a network embedded in the plane, a set of demand points, and a finite candidate set $\mathcal{S}_{cand}$, find a set $S^* \subseteq \mathcal{S}_{cand}$ with minimal cardinality covering all demand points. This problem is NP-hard, also if $V \subseteq \mathcal{S}_{cand}$, see [17]. Let an instance of the discrete stop location problem be given. Determine $\underline{m} := \min\{d(s, s') : s, s' \in \mathcal{S}_{cand}$, and $s, s' \in e$ for some $e \in E\}$ as the closest distance between two candidate locations on the same edge. Note that for $a_0, b_0 \to \infty$ we obtain that $d^{max}_{a_0,b_0,v_0} \to 0$. Hence choose $a_0, b_0$ large enough such that $d^{max}_{a_0,b_0,v_0} \leq \underline{m}$. We claim that a solution to the discrete stop location problem with $|S| \leq K$ exists if and only if a solution $S^*$ to (SL-TT-Cov) exists with $g(S^*) \leq \sum_{e \in E)} \frac{d_e}{v_0} + (|E| + K)\left(\frac{v_0}{2a_0} + \frac{v_0}{2b_0}\right)$.

To see this, note that for $d_{e'} \geq d^{max}_{a_0,b_0,v_0}$ for all $e' \in E(S)$ the objective function of (SL-TT-Cov) reduces to

$$g(S) = \sum_{e' \in E(S)} \left(\frac{d_{e'}}{v_0} + \frac{v_0}{2a_0} + \frac{v_0}{2b_0}\right) = \sum_{e \in E} \frac{d_e}{v_0} + |E(S)|\left(\frac{v_0}{2a_0} + \frac{v_0}{2b_0}\right). \qquad (6)$$

"$\Rightarrow$" Let $S$ be a solution to (SL) with $|S| \leq K$. Then there exists another optimal solution $S^* \subseteq \mathcal{S}_{cand}$. Then $S^*$ is feasible for (SL-TT-Cov) and $d_{e'} \geq \underline{m} \geq d^{max}_{a_0,b_0,v_0}$ for all $e' \in E(S^*)$. Hence $g(S) \leq \sum_{e \in E} \frac{d_e}{v_0} + (|E| + K)\left(\frac{v_0}{2a_0} + \frac{v_0}{2b_0}\right)$.

"$\Leftarrow$" Let $S^*$ be a solution to (SL-TT-Cov) with $g(S^*) \leq \sum_{e \in E} \frac{d_e}{v_0} + (|E| + K)\left(\frac{v_0}{2a_0} + \frac{v_0}{2b_0}\right)$. Again, there exists $S \subseteq \mathcal{S}_{cand}$ with $g(S^*) = g(S)$. Since $d_{e'} \geq \underline{m} \geq d^{max}_{a_0,b_0,v_0}$ for all $e' \in E(S^*)$ we have

$$\sum_{e \in E)} \frac{d_e}{v_0} + (|E| + K)\left(\frac{v_0}{2a_0} + \frac{v_0}{2b_0}\right) \geq g(S) = g(S^*) = \sum_{e \in E} \frac{d_e}{v_0} + |E(S)|\left(\frac{v_0}{2a_0} + \frac{v_0}{2b_0}\right),$$

from which we conclude $|E(S)| \leq |E| + K \Leftrightarrow |S| \leq K$.

◀

**Proof.** (Theorem 12) Let $S^* \subseteq \bigcup_{e \in E, p \in \mathcal{P}} T^e(p)$ be optimal, but $S^* \not\subseteq \mathcal{S}_{cand}$. The goal is to replace each $\tilde{s} \in S^* \setminus \mathcal{S}_{cand}$ by a point in $\mathcal{S}_{cand}$ without loosing feasibility or optimality. To this end, take some $\tilde{s} \in S^* \setminus \mathcal{S}_{cand}$. If $\tilde{s} \in V$, then $\tilde{s}$ can be removed, since the vertices are existing stops. Thus, we can assume that $\tilde{s} \notin V$, i.e. $\tilde{s} = (e, x) \in E$. Now find the following points on edge $e$:

- $s_j = (e, x_j), s_{j+1} = (e, x_{j+1}) \in \mathcal{S}_{cand}$ with $s_j <_e \tilde{s} <_e s_{j+1}$ for two consecutive elements of $\mathcal{S}^e_{cand}$ (if they exist on $e$), and
- $s_{left}, s_{right} \in (S^* \cup V) \cap e$ with $s_{left} <_e \tilde{s} <_e s_{right}$ for the two direct neighbors of $\tilde{s}$ on $e$ (which always exist)

We now investigate the objective function if we move $\tilde{s}$. For all $s$ with $s_{left} = (e, x_{left}) <_e s = (e, x) <_e s_{right} = (e, x_{right})$ the objective function $h(x) := g(S \setminus \tilde{s} \cup \{(e, x)\})$ is given as

$$h(x) = \sum_{e' \in E(S \setminus \tilde{s} \cup \{(e,x)\}} T(d_{e'}) = const + T(x - x_{left}) + T(x_{right} - x)$$

where the constant part is independent of the choice of $s = (e, x)$. As in Lemma 4, $h(x)$ is concave in $x$ on the segment between $s_{left}$ and $s_{right}$. Furthermore, from Lemma 11 we know that $cover_{\mathcal{P}}(\{s\}) \supseteq cover\{\tilde{s}\}$ for all $s = (e, x)$ between $s_j$ and $s_{j+1}$. Now consider the minimization problem

$$\min\{h(x) = T(x - x_{left}) + T(x_{right} - x) : \max\{x_{left}, x_j\} \leq x \leq \min\{x_{right}, x_{j+1}\}\}.$$

Due to the concavity of $h(x)$ we know that an optimal solution $x^* \in \{x_{left}, x_{right}, x_j, x_{j+1}\}$ exists.

1. In case that $x^* = x_j$ or $x^* = x_{j+1}$ we may replace $\tilde{s}$ by $s = (e, x^*) \in \mathcal{S}_{cand}$ and hence obtain a feasible solution with the same objective value.
2. In case that $x^* = x_{left}$ or $x^* = x_{right}$ we may delete $\tilde{s}$ since the new solution is still feasible and has the same objective value.

In both cases, we have reduced the number of points in $S^* \setminus \mathcal{S}_{cand}$. Proceeding with remaining points of $S^*$ which do not belong to $\mathcal{S}_{cand}$ finishes the proof. ◀

**Proof.** (Lemma 14) The sum is obtained since the number of candidate edges on each edge $e$ of the original graph $G$ can be calculated independently by $\binom{n+2}{2}$. ◀

# Evolution and Evaluation of the Penalty Method for Alternative Graphs

**Moritz Kobitzsch, Marcel Radermacher, and Dennis Schieferdecker**

**Karlsruhe Institute of Technology, 76128 Karlsruhe, Germany**
`{kobitzsch,schieferdecker}@kit.edu, marcel.radermacher@student.kit.edu`

**Abstract**

Computing meaningful alternative routes in a road network is a complex problem – already giving a clear definition of a best alternative seems to be impossible. Still, multiple methods [1, 2, 4, 17, 18] describe how to compute reasonable alternative routes, each according to their own quality criteria. Among these methods, the *penalty method* has received much less attention than the via-node or plateaux based approaches. A mayor cause for the lack of interest might be the unavailability of an efficient implementation. In this paper, we take a closer look at the penalty method and extend upon its ideas. We provide the first viable implementation –suitable for interactive use– using dynamic runtime adjustments to perform up to multiple orders of magnitude faster queries than previous implementations. Using our new implementation, we thoroughly evaluate the penalty method for its flaws and benefits.

**1998 ACM Subject Classification** G.2.2 Graph Theory

**Keywords and phrases** Alternatives, Routing, Shortest Paths, Penalties, Parallelization

**Digital Object Identifier** 10.4230/OASIcs.ATMOS.2013.94

## 1 Introduction

Finding shortest paths in a road network is a well studied problem. Modern speed-up techniques can compute routes in a split second. These algorithms are usually based on an asymmetric approach: Exploiting the uniqueness of the shortest path distance, the road network is processed and augmented in advance. This –potentially– time-consuming preprocessing step then allows for fast subsequent queries. The approach inherently assumes a static nature of the input, though.

In contrast to the shortest path, alternative routes do not need to be optimal. Existing techniques to compute alternative routes either avoid speed-up techniques completely [2, 4], or try to relax the computational methods used during preprocessing or at query time [1, 18]. Methods that bypass speed-up techniques are only suitable for quality evaluations or offline usage. Thus, algorithms like the *penalty method* [2] are not explored to their full potential, lacking an efficient implementation.

While the authors of [2] hope for an efficient implementation to be feasible, they fail to provide any details on how to achieve this. In fact, until recently it simply was not possible to deal with the amount dynamic changes to the graph required by the penalty method – even though some techniques already existed that could handle small search spaces quite efficiently [5, 21]. By now, techniques such as *Customizeable Route Planning* [6] are available that can be extended to allow for preprocessing of entire continental networks within less than a second. This (near) real-time processing is achieved by extensive use of parallelism and vectorization and brings an efficient implementation of the penalty method within reach.

In this paper we show how to achieve an efficient implementation of the penalty method, providing speed-ups up to multiple orders of magnitude above previous implementations [2].

Based on this implementation, we thoroughly evaluate the potential of the penalty method. The paper is structured as follows: Our terminology is introduced in Section 2, followed by related work in Section 3. Our contribution to the penalty method is discussed in Section 4. Supporting experiments are presented in Section 5, before a conclusion is drawn in Section 6.

## 2 Preliminaries

Every road network can be viewed as a directed and weighted graph:

▶ **Definition 1** (Graph,Restricted Graph)**.** A weighted **graph** $G = (V, A, c)$ is described as a set of vertices $V, |V| = n$, a set of arcs $A \subseteq V \times V, |A| = m$ and a cost function $c : A \mapsto \mathbb{N}_{>0}$. We might choose to restrict $G$ to a subset $\widetilde{V} \subseteq V$. This **restricted graph** $G_{\widetilde{V}}$ is defined as $G_{\widetilde{V}} = (\widetilde{V}, \widetilde{A}, c)$, with $\widetilde{A} = \{a = (u, v) \in A \mid u, v \in \widetilde{V}\}$.

We define paths and the associated distances as follows:

▶ **Definition 2** (Paths,Length,Distance)**.** Given a graph $G = (V, A, c)$: We call a sequence $p_{s,t} = \langle s = v_0, \ldots, v_k = t\rangle$ with $v_i \in V, (v_i, v_{i+1}) \in A$ a **path** from $s$ to $t$. Its **length** $\mathcal{L}(p_{s,t})$ is given as the combined weights of the represented arcs: $\mathcal{L}(p_{s,t}) = \sum_{i=0}^{k-1} c(v_i, v_{i+1})$. If the length of a path $p_{s,t}$ is minimal over all possible paths between $s$ and $t$ with respect to $c$, we call the path a **shortest path** and denote $p_{s,t} = \mathcal{P}_{s,t}$. The length of such a shortest path is called the **distance** between $s$ and $t$: $\mathcal{D}(s, t) = \mathcal{L}(\mathcal{P}_{s,t})$. Furthermore, we define $\mathcal{P}_{s,v,t}$ as the **concatenated path** $\mathcal{P}_{s,v,t} = \mathcal{P}_{s,v} \cdot \mathcal{P}_{v,t}$.

In the context of multiple graphs or paths, we denote the desired restriction via subscript. For example $\mathcal{D}_G(s, t)$ denotes the distance between $s$ and $t$ in $G$, with $\mathcal{D}_{p_{s,t}}(a, b)$ we denote the distance between $a$ and $b$ when following $p_{s,t}$.

Our metric of choice is the average travel time. Therefore, we might choose to omit the cost function $c$ when naming a graph and simply write $G = (V, A)$.

## 3 Related Work

Both shortest paths and alternative routes computation are important for our work. Following, we give a short overview of the techniques most relevant to our contribution.

### Shortest Paths

Algorithms for computing exact shortest paths have come a long way. Starting back in the late 1950s with Dijkstra's algorithm [12], incredible progress was made in this area – especially during the last decade. By now, we can answer distance queries on a road network over a million times faster than Dijkstra's algorithm.

All relevant *speed-up techniques* to Dijkstra's algorithm share an asymmetric approach: In a preprocessing step, auxiliary information is generated once and then used in all subsequent queries. This approach is effective if arc costs do not change, but it becomes a major bottleneck if not prohibitive, if preprocessing has to be repeated multiple times. [3,9,23] give a general overview on speed-up techniques as we focus on the following two techniques:

*Contraction Hierarchies* [14] (CH) is probably the most studied speed-up technique. During preprocessing, the graph representing the road network is augmented by carefully chosen (shortcut) arcs while arcs not required for correctness are removed. This results in a sparse directed acyclic graph (DAG). A query corresponds to a bidirectional variant of Dijkstra's algorithm on this modified graph. Road networks of continental size can be preprocessed within minutes and distance queries run in the order of one hundred microseconds. Reconstructing the complete shortest path requires roughly the same time. This technique is most suited for static settings in which graphs do not change.

*Customizeable Route Planning* [6] (CRP, also known as Multi Level Dijkstra) is the current pinnacle in a long list of multi-level separator based techniques, [8, 16, 22] to name a few. In a first preprocessing step, a multi-level partition is generated. The boundary vertices of this partition induce an overlay graph at each level. To maintain shortest paths, each cell is connected in a clique. Computing this representation relies only on the structure of the graph. A metric is incorporated in a second step, when correct costs are computed for all arcs within the cliques. The (bidirectional) query traverses arcs like Dijkstra's algorithm. When a boundary vertex is reached, the query switches to the next higher level and continues to traverse only arcs in the respective overlay graph[1]. CRP profits from using *PUNCH* [7] to find tiny separators. The best variant uses a combination of up to 5 levels and an additional set of guidance levels (or *shadow levels*) for preprocessing. This setup allows for distance queries in about one millisecond and updates of the entire metric in less than a second.

## Alternative Routes

Abraham et al. [1] were the first to formally introduce alternative routes in road networks, even though related methods like the k-shortest path problem [13, 24] have been introduced before. We do not cover these methods as no suitable implementation exists for continental sized road networks and because some of the earlier methods do not produce good alternative routes due to a plethora of short detours available in road networks. By now, two most common approaches found in the literature are via-node alternative routes or the related plateaux method [1, 2, 4, 17, 18], and penalty-based approaches [2] (among others). Their following description is taken partly from [17]:

### Via-Node Alternative Routes

Within a graph $G = (V, A)$, a via-node alternative to a shortest path $\mathcal{P}_{s,t}$ can be described by a single vertex $v \in V \setminus \mathcal{P}_{s,t}$. The alternative route is described as $\mathcal{P}_{s,v,t}$. As this simple description can result in arbitrarily bad paths, for example paths containing loops, Abraham et al. [1] define a set of criteria to be fulfilled for an alternative route to be *viable*. A viable alternative route provides the user with a real alternative, not just minimal variations (*limited sharing*), is not too much longer (*uniformly bounded stretch*) and does not contain obvious flaws, i.e. sufficiently small sub-paths have to be optimal (*local optimality*). Formally, these criteria are defined as follows:

▶ **Definition 3** (Viable Alternative Route). Given a graph $G = (V, A)$, a source $s$, a target $t$, and a via-node $v$ as well as three tuning parameters $\gamma, \epsilon, \alpha$; $v$ is a viable via-node and defines a **viable via-node alternative route** $\mathcal{P}_{s,v,t} = \mathcal{P}_{s,v} \cdot \mathcal{P}_{v,t}$, if following criteria are fulfilled:

1. $\mathcal{L}(\mathcal{P}_{s,t} \cap \mathcal{P}_{s,v,t}) \leq \gamma \cdot \mathcal{D}(s,t)$ *(limited sharing)*
2. $\forall a, b \in \mathcal{P}_{s,v,t},\ \mathcal{D}_{\mathcal{P}_{s,v,t}}(s,a) < \mathcal{D}_{\mathcal{P}_{s,v,t}}(s,b):$
   $\mathcal{D}_{\mathcal{P}_{s,v,t}}(a,b) \leq (1+\epsilon) \cdot \mathcal{D}(a,b)$ *(uniformly bounded stretch)*
3. $\forall a, b \in \mathcal{P}_{s,v,t},\ \mathcal{D}_{\mathcal{P}_{s,v,t}}(s,a) < \mathcal{D}_{\mathcal{P}_{s,v,t}}(s,b),$
   $\mathcal{D}_{\mathcal{P}_{s,v,t}}(a,b) \leq \alpha \cdot \mathcal{D}(s,t):\ \mathcal{D}_{\mathcal{P}_{s,v,t}}(a,b) = \mathcal{D}(a,b)$ *(local optimality)*

[1] For efficiency, the query may descend to lower levels when close to target/source. See [6] for details.

The usual choice for the tuning parameters is to allow for at most $\gamma = 80$ % overlap between $\mathcal{P}_{s,v,t}$ and $\mathcal{P}_{s,t}$. Furthermore the user should never travel more than $\epsilon = 25$ % longer than necessary between any two points on the track, and every subpath that is at most $\alpha = 25$ % as long as the original shortest path should be an optimal path.

These criteria require a quadratic number of shortest path queries to be fully evaluated. Therefore, Abraham et al. propose an approximation [1]. For example, their *T-test* for local optimality is achieved by a single query between two vertices close to the via-node. Due to properties of this T-test, the criteria (and thus also the numbers in Section 5) are usually evaluated only for the part of the via-route that is distinct from the shortest path.

Definition 3 can be directly extended to allow for a second or third alternative (alternative routes of *degree* 2, 3 or even $n$), as only the limited sharing parameter has to be tested against the full set of alternatives already known.

Abraham et al. [1] give multiple algorithms to compute alternative routes. The reference algorithm (X-BDV) is based on a bidirectional implementation of Dijkstra's algorithm and is used as the gold standard. To avoid the long query time of Dijkstra's algorithm on continental-sized networks, they also give techniques based on Reach [15] and Contraction Hierarchies [14]. Due to the strong restrictions of the search spaces caused by the speed-up techniques, they present weakened query criteria which they call relaxation. For example, in the Contraction Hierarchy they allow to look downwards in the hierarchy under certain conditions. The relaxation can be applied in multiple intensities. Commonly used is the 3-relaxation which we refer to by X-CHV.

Luxen and Schieferdecker [18] improve the algorithm of Abraham et al. [1] in terms of query times by storing a precomputed small set of via-nodes for pairs of regions within the graph. This is the fastest method to compute via-node alternative routes as of now.

## The Penalty Method

Bader et al. [2] describe an entirely different approach. Their main focus is on computing a full alternative graph to present as a whole, or possibly to extract alternative routes from. The construction of the graph, however, relies on the iterative computation of multiple shortest paths. After a single path is computed, they apply a penalty to the path and to every arc directly connected to it, thus potentially finding a different path in the next iteration. By lowering successive increases in penalty, they propose to stop iterating when no penalty is applied to the extracted path anymore. The publication itself does not give exact numbers, but according to one of the authors 20 iterations are performed to generate paths. From this set of computed paths, the best ones are selected and combined into an alternative graph. To make the graph more readable, the authors present two filters that can be applied to the graph. Again, they do not specify any details on how to select the paths for the final graph. The quality of the graph is evaluated using two measures (total and average distance) while at the same time limiting the complexity by putting a hard bound on vertices of degree higher than two (*decision vertices*). It is defined as follows:

▶ **Definition 4** (Alternative Graph Quality). The quality of an **alternative graph** $H = (V_H, A_H)$, also called *target function*, is given by $\mathit{totalDistance} - (\mathit{averageDistance} - 1)$ with $\mathit{averageDistance} \in [1.0, 1.1]$. The upper limit is enforced during graph construction. Given start $s$ and target $t$, the contributing values are defined as:

1. $\mathit{totalDistance} := \sum_{a=(u,v)\in A_H} \frac{c(a)}{\mathcal{D}_H(s,u)+c(a)+\mathcal{D}_H(v,t)}$ *(indicates sharing)*
2. $\mathit{averageDistance} := \frac{\sum_{a\in A_H} c(a)}{\mathcal{D}(s,t)\cdot \mathit{totalDistance}}$ *(indicates stretch)*

Intuitively speaking, the total distance describes how many distinct paths can be found in the graph, while the average distance describes how much longer such a path is on average. The authors themselves do not provide an efficient implementation of the penalty method but claim the implementation to be possible with [21]. With single arc updates taking several milliseconds this claim seems excessive.

Recently, Paraskevopoulos and Zaroliagis published a new paper [19] on the penalty method. They suggest modifications to the penalization scheme to obtain higher quality alternative graphs. Additionally, they introduce pruning techniques that allow for faster query times than the original work by Bader et al. [2].

## 4 Alternative Graph Computation

The basic setup of our algorithm follows the ideas from [2], as described in Section 3. We compute a shortest path, potentially add the obtained path to our output, penalize the arcs on the path as well as the adjoined arcs and repeat. The general process is illustrated below as Algorithm 1:

**Algorithm 1** CRP-$\pi$

```
$original\_path$ = $path$ = computeShortestPath();
$H$ = $original\_path$;
while $\mathcal{L}(path) \leq (1+\epsilon) \cdot \mathcal{L}(original\_path)$ do
begin
    applyPenalties($path$);
    $path$ = computeShortestPath();
    if $isFeasable(path)$ do
    begin
        $H$ = $H \cup path$;
    end;
end;
return $H$;
```

Following, we explain the meaning of `isFeasable(path)` and the stopping criterion of our algorithm. We describe the modifications we make to the algorithm proposed by Bader et al., which we also refer to as *classcial* method. We show how to achieve an efficient implementation and how to extract single alternative routes.

### 4.1 Path Selection

Algorithm 1 utilizes the procedure `isFeasible(path)` to decide whether to keep an alternative route or not. The original paper [2] does not specify how to exactly choose the paths that form the alternative graph. According to personal conversation with one of the authors, their algorithm computes up to 20 paths and performs a selection based on some priority terms afterwards.

While their implementation does not focus on query times but on evaluating multiple different approaches, we have to consider the cost of performing too many iterations. Thus, we take a different approach. Instead of applying penalties as many as 20 times, we consider the true length –without penalties– of the computed path. Whenever this length exceeds $(1+\epsilon) \cdot \mathcal{D}(s,t)$, we stop our algorithm (see Definition 3, uniformly bounded stretch).

Every path we find during the execution is evaluated by the aforenamed procedure for its potential *value* to the alternative graph. We postulate that a path must offer at least one deviation to the current alternative graph of length $\delta \cdot \mathcal{D}(s,t)$ or more, with $\delta$ usually chosen as $\delta = 0.1$ (compare Definition 3, limited sharing). The detours satisfying this requirement are checked against the current alternative graph $H$ for stretch. If one of these detours between vertices $a$, $b$ is not longer than $(1+\epsilon)\cdot\mathcal{D}_H(a,b)$, its containing path is added to the alternative graph. All other paths are rejected.

## 4.2 Changes to Penalization

As described above, the classical penalty method penalizes the arcs along the shortest path by adding a fraction of the arcs' original length. This fraction is called the *penalty factor* $\pi_f$ and is usually chosen as $\pi_f = 0.04$. Additionally, the adjoined arcs of the path are penalized by adding $\pi_r = \pi_f \cdot 0.002 \cdot \mathcal{D}(s,t)$, the so called *rejoin penalty.* These changes to the metric are persistent during the computation of an alternative graph.

The penalty method, as suggested by Bader et al. [2], requires a significant amount of iterations. Even when using our stopping criterion from the previous section, we experience a similar behavior, requiring about 20 iterations and more on average. Therefore, we take a slightly different approach to penalization and modify the way penalties are introduced to the graph.

In contrast to the choice of Bader et al., we multiply the current lengths of the arcs on the shortest path by $1+\pi_f$. Thus, we penalize arcs which are used often more strongly than others, and significantly increase the rate at which penalties grow (geometrically growing). This is in contradiction to Bader et al.'s preference to lower successive increases in penalty. Our choice is not only motivated by its beneficial impact on the number of iterations, but also favors detours to segments of the alternative graph that are already covered by multiple paths. But of course, this may result in our algorithm missing some promising path candidates due to the higher increase in penalties.

The higher penalization has influence on the choice of the rejoin penalty as well. While Bader et al. propose a relatively small penalty $\pi_r$, this choice proved to be not high enough for our faster growing multiplicative penalties to prevent short detours. We therefore change the additive penalty to $\pi_r = \alpha \cdot \sqrt{\mathcal{D}(s,t)}$, with $\alpha = 0.5$ as a typical value. This change is motivated by the following observation: Consider $\mathcal{P}_{s,t}$ and a new path $p_{s,t}$ found on the penalized graph which deviates from $\mathcal{P}_{s,t}$ between vertices $a, b$. Due to penalization, $\mathcal{D}_{p_{s,t}}(a,b) + 2\cdot\pi_r \leq (1+\bar{\pi})\cdot\mathcal{D}(a,b)$, with $\bar{\pi}$ the average penalization along $\mathcal{P}_{a,b}$. In other words, for a new detour to be found between vertices $a, b$, it has to be shorter, including rejoin penalties, than the current (penalized) shortest path between $a, b$. This condition gets easier to fulfill, the further $s, t$ are apart as the rejoin penalty grows much slower than the path length. Therefore, we allow for larger detours to be found on longer paths. But locally, we only want short detours (compare Definition 3, local optimality).

## 4.3 Fast Computation

Our most significant change to the classical method is the introduction of CRP as speed-up technique. While there are other methods for dynamic shortest path computation [5, 21], none of them is sufficient for the high amount of dynamic behavior required for our cause. As recent developments have shown, close to real-time processing of entire graphs is possible with the CRP technique [10].

While we initially took a different approach, the general methods –making extensive use of vector instructions and parallelism– remain the same. Even though it does not seem intuitive to perform more work than necessary for a pure update of a shortest path, the locality properties, vectorization and suitability for parallel processing allow a CRP based implementation to outperform other approaches that consider only the changed arcs.

As our set of updating techniques –while much simpler to implement– is outperformed by the methods recently described by Delling and Werneck [10], we do not focus on our exact implementation of CRP. Instead, we concentrate on an essential modification to CRP, required to achieve maximal performance of the penalty method:

### Dynamic Level Selection

Applying a multi-level technique is always a balancing act between fast queries of long routes and overhead for short routes. In our case, namely the reiterated computation of a shortest path and the update of affected arcs between the same source and target, we also have to weigh update costs against query times. While beneficial for long range queries, large cells in the upper levels have a high update cost in comparison to the smaller cells further down the hierarchy. These updates soon dominate the runtime of our algorithm as even rather short paths can touch many high level cells. As we perform multiple computations between the same source and target pair, we can alleviate this problem. After an initial computation of the shortest path, we can analyze the path regarding its length and the cells the path touches on different hierarchy levels. For the cost of storing some additional mapping information, we use the obtained information to dynamically adjust our implementation of CRP to work only on a fixed number of levels. Restricting to a subset of levels allows for faster updates of CRP as not all levels have to be updated, at the cost of higher query times.

## 4.4 Alternative Route Extraction

When presented with an alternative graph, multiple ways exist to extract alternative routes. To compare ourselves to other methods in terms of success rate, we apply a two-step approach. In a first step, we perform X-BDV on our alternative graph without checking for local optimality. This test is omitted as the alternative graph does not provide enough information to compute shortest paths between arbitrary vertices with respect to the underlying graph. We call the result of this first step *CRP-$\pi$-via*. After having searched exhaustively for via-node alternatives, we run a simplified penalty method to extract further routes. We do not apply rejoin penalties as the alternative graph is sparse and only contains meaningful junctions. The X-BDV alternatives are used to initialize penalization, and the extracted paths have to adhere to the same stretch and overlap criteria as imposed by X-BDV. The full algorithm is named *CRP-$\pi$*. When omitting the first step, we call the results *CRP-$\pi$-penalty*.

## 5 Experiments

We first provide a detailed overview of our experimental setup and a short outline of the quality measurements used during the evaluation of the penalty method. This is followed by an extensive experimental evaluation of our techniques as introduced in Section 4.

### Setup

We run our algorithm on four Intel Xeon E5-4640, clocked at 2.4 GHz with 32 cores in total and 512 GB of RAM – the actual space consumption of CRP-$\pi$ is much less though. The machine is running Ubuntu 12.04. We apply the C++ compiler of the GNU Compiler Collection (GCC), version 4.6.1, with parameters `-std=c++0x -fopenmp -O3 -msse4.1 -mtune=native`. For parallelization we use OpenMP. Our implementation is based on a partition generated by PUNCH, comparable to the one used in [6] with 5 levels, including a shadow level. The road network of Western Europe supplied by PTV AG for the 9th Dimacs Implementation Challenge [11] is used in our experiments. It contains 18 million nodes and 24 million edges and uses the travel time metric as arc costs. The graph consists of a single strongly connecteed component. We present numbers based on random queries and on rank queries. The Dijkstra rank of a vertex is defined as the step in which it is settled during the execution of Dijkstra's algorithm. For both variants we choose 1 000 queries uniformly at random, unless said otherwise.

For comparison, we apply our own implementations of the competing via-node and penalty approaches, with results similar to the original papers. Tuning parameters are chosen at their usual values as introduced before. Note that our implementation of the classical penalty method corresponds to CRP-$\pi$ while using Dijkstra's algorithm and the classical penalization scheme. In particular, we apply our stopping criterion instead of running 20 iterations straight and add each viable path to the alternative graph as soon as it is discovered. This is due to Bader et al. not providing details on the path selection process in their paper.

## 5.1 Runtime

One of the most important characteristics in routing applications is the query time. Therefore, we first evaluate the runtime of our algorithm before turning to the quality of the computed routes.

The number of iterations performed is likely the most influential factor to our query time. Figure 1 shows that the number of iterations is much higher for queries of lower rank. Therefore, it is important to perform updates very fast for short queries. We also see that without our modifications to the penalization, we experience an average of 12 iterations and higher across the full range of queries. Remember, the original implementation by Bader et al. always computed 20 iterations. Considering the cost to perform a single iteration (Figure 2), the classical penalization would not be suitable for an efficient implementation, even when using CRP. We also see that update costs remain small as only affected cells are recomputed.

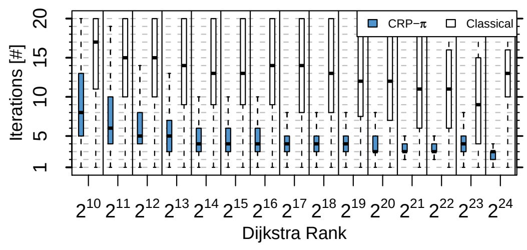


**Figure 1** Number of iterations required by the penalty method until the stopping criterion holds.

These findings are further confirmed by Figure 3, which gives sequential runtimes for using Dijkstra's algorithm with classical penalization. Note the logarithmic scale of the y-axis. We see that query times already become impractical for very short ranged queries. This is expected behavior and comparable to the original algorithm [2].

Using a bidirectional implementaion of Dijkstra's algorithm could reduce runtimes of the classical approach by a small constant factor, but they would remain prohibitively high. This becomes evident in particular when looking at long range queries, the classical approach requires up to 5 seconds per iteration while CRP-$\pi$ only takes between 100 and 200 milliseconds. Not even goal directed methods as in the new method of Paraskevopoulos and Zaroliagis [19] seem to suffice for reaching the query performance of CRP-$\pi$.

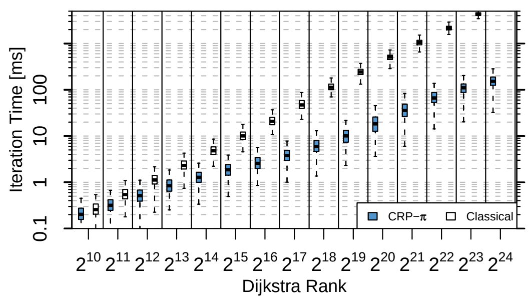


**Figure 2** Sequential runtime of a single iteration of the penalty method.

## Dynamic Level Selection

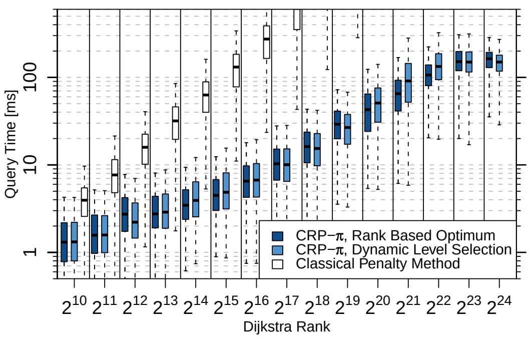


**Figure 3** Runtime of the penalty method with dynamic level selection and rank based optima. CRP-$\pi$ uses 16 cores, the classical method 1 core.

Figure 4 gives an impression on the impact of both parallel execution of our algorithm as well as the overhead introduced by using a fixed number of levels for CRP. Especially for short routes, the negative effects of using many levels of the partition become obvious. This is as the cells in the higher levels are very large and the costly updates dominate the benefits in query time. Using all 5 levels is actually only beneficial in CRP-$\pi$ for finding shortest paths in the original graph. Moreover, we see that using more than 12 cores is hardly effective for any type of query. In fact, parallel execution only starts to pay off for long range queries, with 12/16 cores performing up to 4 times faster than sequential execution. For short to medium range routes, it can be deemed a waste of resources.

Following the results of Figure 4, we propose to use only two levels for short routes, switch to three levels for the medium range, and finally apply four levels on long range queries. We base the decision on how many levels to use on the hop count of the shortest path (which is always computed using all 5 levels of the partition). To tune this selection process, we generate a different set of routes for every Dijkstra rank and compute the average number of hops on the shortest path.

The performance of the resulting dynamic algorithm is shown in Figure 3. We see that our dynamic algorithm is sometimes even slightly faster than the best values of the respective Dijkstra rank queries. This is, as the dynamic choice allows us to better adjust the performance of our algorithm to the actually required workload whereas a forced number of levels does not represent the required work as accurately. For comparison, Figure 3 shows the respective best values from Figure 4. The values were extracted on Dijkstra-rank basis. Due to very long arcs, i.e. ferry connections, we experience some erratic parallelization behavior when dynamically selecting the levels based on the hop-count alone. A selection based on affected cells in each level might provide better results in the future.

Avoiding dynamic level selection, the best choice is obviously using three levels of hierarchy. Only for very short and very long range queries we see detrimental effects. But in the worst case, there is a slowdown by a factor of up to 4.

Although, at the current state, the required number of cores for a viable execution of our algorithm might be considered high for long range queries, the algorithm performs queries efficiently and with a low number of cores for all reasonable distances. To improve workload, dynamic selection of cores can be introduced similar to dynamic selection of levels.

Figure 3 compares only our best results, obtained by parallel processing, to the classical approach, running sequentially. Though, we clearly outperform this approach even when using a single core. Figure 4 demonstrates the performance benefits of our general approach over the classicial method. As seen in the plot depicting dynamic level selection, CRP-$\pi$ takes at most 600 milliseconds on one core for the longest queries whereas the classical approach requires beyond 100 seconds, which is off the scale in Figure 3.

For completeness, we state the runtimes of the via-node approaches introduced during the following quality analysis. X-BDV requires 14.1 seconds on average to compute three alternatives, where possible. X-CHV takes 4.95 milliseconds for the same task.

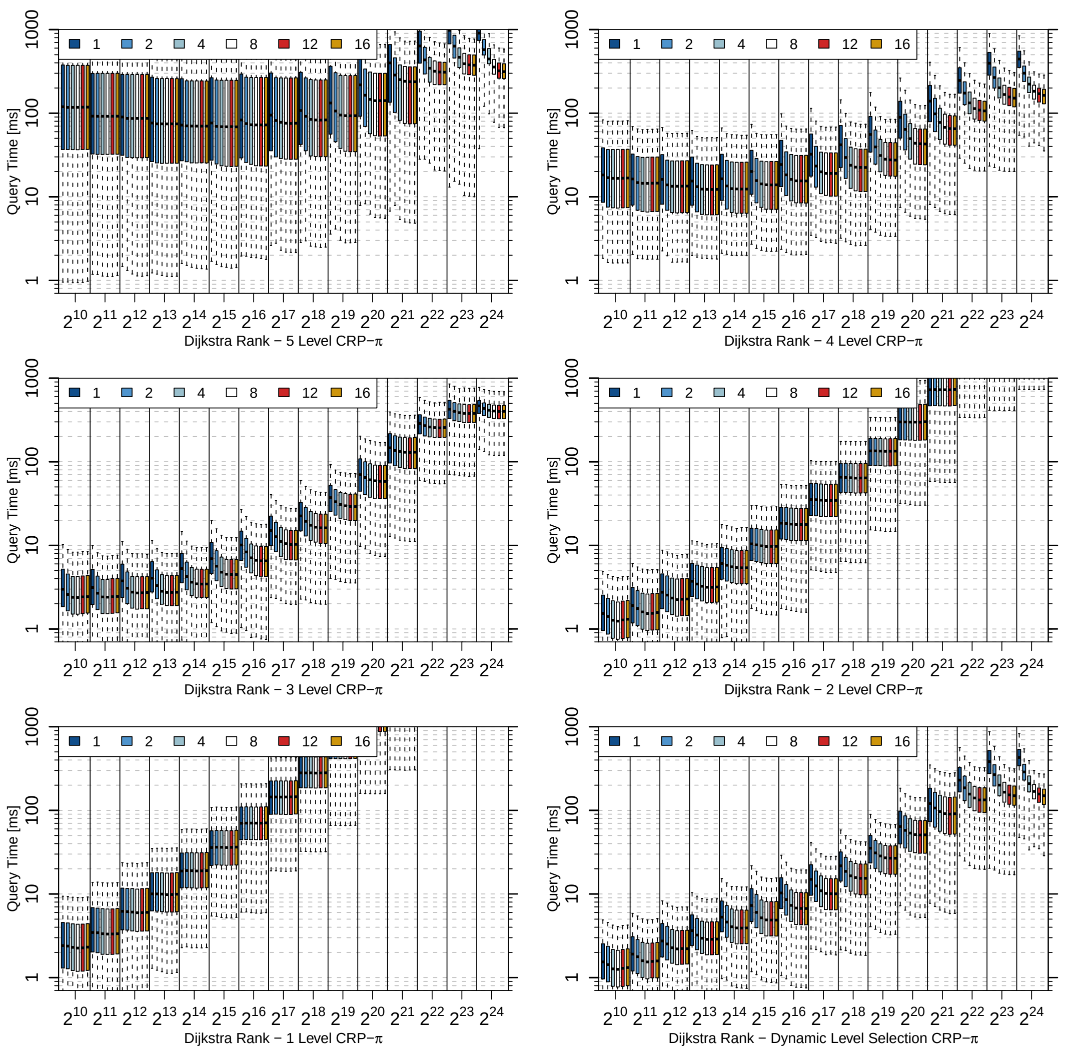


**Figure 4** Runtime of the penalty method. Each illustration shows the runtime for a range of cores. The number of levels of the underlying CRP implementation is either fixed to a given set of levels, or dynamically adapted after the initial shortest path query.

## 5.2 Quality

First, we evaluate the results of our algorithm with regards to the quality as defined in the original paper of Bader et al. [2]. We note that without access to their path selection criteria, we were unable to reproduce the numbers listed in their work. In addition, their numbers stem from a very sparse test set of 100 queries. Another implementation [20] faced similar difficulties, but at least conducted more extensive measurements. They report an average value of 2.89 for the alternative graph quality of Definition 4, compared to the 3.21 in [2]. Without filtering, our algorithm yields alternative graphs with an average quality rating of 3.32 with 17.4 decision vertices on average. As this is above the proposed hard limit in the original paper, we can filter the graph by removing all arcs that are only contained in paths longer than the allowed maximum stretch. The additional overhead is negligible, well below 100$\mu$s on average, as the alternative graphs are tiny. Filtering reduces the quality to 2.89 and the decision vertices to 9.53. While limiting the number of decision vertices is beneficial

**Table 1** Success rates and average path quality numbers for the first through third alternatives in terms of Definition 3. Compared to the results found in [1], local optimality is not strictly enforced. See text for a discussion on maximum/minimum path quality values.

| # | algorithm | success [%] | stretch [%] | sharing [%] | optimality [%] |
|---|---|---|---|---|---|
| first | X-BDV | 96.0 | 10.0 | 41.8 | 75.4 |
| | X-CHV | 89.6 | 80.4 | 40.6 | 68.1 |
| | CRP-$\pi$-via | 95.2 | 42.8 | 31.6 | 27.1 |
| | CRP-$\pi$-penalty | 96.3 | 40.6 | 40.8 | 24.4 |
| | CRP-$\pi$ | 96.3 | 42.9 | 31.9 | 26.9 |
| second | X-BDV | 87.6 | 13.8 | 59.5 | 65.1 |
| | X-CHV | 72.5 | 269.0 | 57.6 | 57.2 |
| | CRP-$\pi$-via | 79.8 | 47.1 | 44.7 | 22.9 |
| | CRP-$\pi$-penalty | 83.1 | 60.5 | 36.8 | 10.8 |
| | CRP-$\pi$ | 84.0 | 47.6 | 45.9 | 22.1 |
| third | X-BDV | 75.5 | 17.2 | 65.6 | 54.6 |
| | X-CHV | 51.4 | 214.0 | 63.6 | 46.8 |
| | CRP-$\pi$-via | 52.7 | 66.5 | 49.3 | 18.0 |
| | CRP-$\pi$-penalty | 53.0 | 65.9 | 32.0 | 5.6 |
| | CRP-$\pi$ | 62.9 | 67.4 | 51.8 | 15.9 |

for the readability of an alternative graph, it reduces the potential for extracting multiple viable alternatives. Thus, we opt to not apply the filter for the following analyses.

Now, we take a look at the well established via-node approach. For comparison, we choose the Dijkstra based (X-BDV), and the Contraction Hierarchy based (X-CHV) variants introduced in [1]. Table 1 summarizes the results, giving numbers for success rates, i.e. how often we can extract one to three viable alternatives from our alternative graph, and for the quality measures introduced in Definition 3. As we do not strictly enforce local optimality, we disabled this criterion for X-BDV and X-CHV to allow for a fair comparison. We see that success rates of CRP-$\pi$ are well above X-CHV for all alternatives and even on par with X-BDV for the first route. For second and third routes our algorithm fares slighty worse compared to X-BDV. Note though that X-BDV obtains its high success rates at the cost of prohibitively slow query times of about 14 seconds. Average path quality measures seem reasonable with our uniformly bounded stretch and local optimality values being worse but our sharing values being better than those of the via-node approaches. This is an expected compromise as lower stretch comes with higher overlap and vice versa. The overall high stretch values are due to none of the algorithms enforcing uniformly bounded stretch explicitly, only total stretch of each path is enforced.

We further find that not enforcing local optimality has little impact on the average path quality values. Only the uniformly bounded stretch of X-CHV increases significantly. Due to the structure of CH search spaces, computed alternatives often exhibit an overlapping subpath at the via-node. This would imply infinite stretch values, filtering these overlaps we obtain the listed values of above 200. Maximum stretch and minimum optimality values degrade dramatically without enforcing local optimality, though. This leads us to look more closely into how poor bounded stretch and local optimality values arise. As they represent averages of the worst values on each path, it is easy to see that even tiny suboptimalities compared to the full path length lead to poor quality values. We find stretch values over

100% for about 10% of all alternatives and local optimality values below 1% for about 20% of all alternatives of CRP-$\pi$. This is about twice as often as for X-CHV, with X-BDV only showing single poor values. We further checked that this is always caused by a single subpath of small length (about 1% - 3% of the full path length). Thus, we conjecture that these problems are repairable with very local searches at low costs.

Finally, we compare the results of running only one step of our alternative route extraction to the full process. We see that the different extraction methods compliment each other. On their own, they offer comparable success rates on par or even better than X-CHV. When combined, the success rates increase – especially for higher degree alternatives. This leads us to the conjecture that both approaches provide structural different routes.

Encouraged by this observation on the small scale, we study the structural differences offered by our penalty approach compared to via-node based approaches on the whole. We evaluate the extracted alternative routes with regards to the additional information they provide that cannot be obtained by via-node based approaches. For this analysis we consider all extracted routes $p_{s,t}$, compute a via-node alternative $\mathcal{P}_{s,v,t}$ for each vertex $v$ on that route, and compute the overlap between these two paths. For a fair comparison, we only consider vertices that yield a viable via-node alternative with respect to the stretch and overlap criteria. Furthermore, we do not consider overlapping subpaths that are also part of the shortest path $\mathcal{P}_{s,t}$. We find that that maximum overlap is well below 80% and getting smaller for higher degree alternatives – 77.9%, 72.7%, 65.5% for the first through third alternative, respectively. This implies that our approach offers a meaningful addition to the established via-node approaches.

## 6 Conclusion

The extensive use of vectorization and modern multi-core machines has enabled us to provide the first implementation of the penalty method that is suitable for interactive applications. We have shown the results to provide meaningful additions to the world of alternative routes. Some open problems remain though. Most interesting would be to find an (approximable) set of criteria to classify good alternatives, not tailored to one specific approach. Furthermore, the runtime of our implementation remains high, especially when compared to shortest path queries. We want to find ways to improve upon this implementation and compute alternative graphs even quicker. The recent work by Paraskevopoulos and Zaroliagis [19] seems to be promising in this respect. Their approach is orthogonal to ours and should integrate well.

**Acknowledgements** We would like to thank Daniel Delling of Microsoft Reasearch, Sillicon Valley for providing a PUNCH partition of our European road network.

## References

1 Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. Alternative Routes in Road Networks. *ACM Journal of Experimental Algorithmics*, 18(1):1–17, 2013.

2 Roland Bader, Jonathan Dees, Robert Geisberger, and Peter Sanders. Alternative Route Graphs in Road Networks. In *International ICST Conference on Theory and Practice of Algorithms in (Computer) Systems (TAPAS'11)*, volume 6595 of *LNCS*, pages 21–32. Springer, 2011.

3 Reinhard Bauer, Daniel Delling, Peter Sanders, Dennis Schieferdecker, Dominik Schultes, and Dorothea Wagner. Combining Hierarchical and Goal-Directed Speed-Up Techniques for Dijkstra's Algorithm. *ACM Journal of Experimental Algorithmics*, 15(2.3):1–31, 2010.

4 Cambridge Vehicle Information Tech. Ltd. Choice Routing. `http://camvit.com/camvit-technical-english/Camvit-Choice-Routing-Explanation-english.pdf`, 2005.

5 Gianlorenzo D'Angelo, Mattia D'Emidio, Daniele Frigioni, and Camillo Vitale. Fully Dynamic Maintenance of Arc-Flags in Road Networks. In *International Symposium on Experimental Algorithms (SEA'12)*, volume 7276 of *LNCS*, pages 135–147. Springer, 2012.

6 Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. Customizable Route Planning. In *International Symposium on Experimental Algorithms (SEA'11)*, volume 6630 of *LNCS*, pages 376–387. Springer, 2011.

7 Daniel Delling, Andrew V. Goldberg, Ilya Razenshteyn, and Renato F. Werneck. Graph Partitioning with Natural Cuts. In *International Symposium on Parallel and Distributed Processing (IPDPS'11)*, pages 1135–1146. IEEE, 2011.

8 Daniel Delling, Martin Holzer, Kirill Müller, Frank Schulz, and Dorothea Wagner. High-Performance Multi-Level Routing. In *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74 of *DIMACS Book*, pages 73–92. AMS, 2009.

9 Daniel Delling, Peter Sanders, Dominik Schultes, and Dorothea Wagner. Engineering Route Planning Algorithms. In *Algorithmics of Large and Complex Networks*, volume 5515 of *LNCS*, pages 117–139. Springer, 2009.

10 Daniel Delling and Renato F. Werneck. Faster Customization of Road Networks. In *International Symposium on Experimental Algorithms (SEA'13)*, volume 7933 of *Lecture Notes in Computer Science*, pages 30–42. Springer, 2013.

11 Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson, editors. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74 of *DIMACS Book*. AMS, 2009.

12 Edsger W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1:269–271, 1959.

13 David Eppstein. Finding the $k$ Shortest Paths. In *Symposium on Foundations of Computer Science (FOCS'94)*, pages 154–165. IEEE, 1994.

14 Robert Geisberger, Peter Sanders, Dominik Schultes, and Christian Vetter. Exact Routing in Large Road Networks Using Contraction Hierarchies. *Transportation Science*, 46(3):388–404, 2012.

15 Ronald J. Gutman. Reach-Based Routing: A New Approach to Shortest Path Algorithms Optimized for Road Networks. In *Workshop on Algorithm Engineering and Experiments (ALENEX'04)*, pages 100–111. SIAM, 2004.

16 Martin Holzer, Frank Schulz, and Dorothea Wagner. Engineering Multi-Level Overlay Graphs for Shortest-Path Queries. *ACM Journal of Experimental Algorithmics*, 13(2.5):1–26, 2008.

17 Moritz Kobitzsch. An Alternative Approach to Alternative Routes: HiDAR. In *European Symposium on Algorithms (ESA'13)*. Springer, 2013.

18 Dennis Luxen and Dennis Schieferdecker. Candidate Sets for Alternative Routes in Road Networks. In *International Symposium on Experimental Algorithms (SEA'12)*, volume 7276 of *LNCS*, pages 260–270. Springer, 2012.

19 Andreas Paraskevopoulos and Christos Zaroliagis. Improved Alternative Route Planning. In *Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS'13)*, OASIcs. Dagstuhl Publishing, 2013.

20 Marcel Radermacher. Schnelle Berechnung von Alternativgraphen. Bachelor's thesis, Karlsruhe Institute of Technology, Fakultät für Informatik, 2012.

21 Dominik Schultes and Peter Sanders. Dynamic Highway-Node Routing. In *Workshop on Experimental Algorithms (WEA'07)*, volume 4525 of *LNCS*, pages 66–79. Springer, 2007.

22 Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. Using Multi-Level Graphs for Timetable Information in Railway Systems. In *Workshop on Algorithm Engineering and Experiments (ALENEX'02)*, volume 2409 of *LNCS*, pages 43–59. Springer, 2002.
23 Christian Sommer. Shortest-Path Queries in Static Networks, 2012. submitted. Preprint available at `http://www.sommer.jp/spq-survey.htm`.
24 Jin Y. Yen. Finding the $k$ Shortest Loopless Paths in a Network. *Management Science*, 17(11):712–716, 1971.

# Improved Alternative Route Planning *

**Andreas Paraskevopoulos[1,2] and Christos Zaroliagis[1,2]**

1 **Computer Technology Institute & Press "Diophantus"**
**Patras University Campus, 26504 Patras, Greece**
2 **Department of Computer Engineering & Informatics**
**University of Patras, 26504 Patras, Greece**
`{paraskevop,zaro}@ceid.upatras.gr`

## Abstract

We present improved methods for computing a set of alternative source-to-destination routes in road networks in the form of an alternative graph. The resulting alternative graphs are characterized by minimum path overlap, small stretch factor, as well as low size and complexity. Our approach improves upon a previous one by introducing a new pruning stage preceding any other heuristic method and by introducing a new filtering and fine-tuning of two existing methods. Our accompanying experimental study shows that the entire alternative graph can be computed pretty fast even in continental size networks.

**1998 ACM Subject Classification** F.2.2 Nonnumerical Algorithms and Problems, G2.2.2 Graph Theory, G.4 Mathematical Software

**Keywords and phrases** Alternative route, stretch factor, shortest path, non-overlapping path, Penalty, Plateau

**Digital Object Identifier** 10.4230/OASIcs.ATMOS.2013.108

## 1 Introduction

Route planning services – offered by web-based, hand-held, or in-car navigation systems – are heavily used by more and more people. Typically, such systems (as well as the vast majority of route planning algorithms) offer a best route from an origin $s$ to a destination $t$, under a single criterion (distance or time). Quite often, however, computing only one such $s$-$t$ route may not be sufficient, since humans would like to have choices and every human has also his/her own preferences. These preferences may well vary and depend on specialized knowledge or subjective criteria (like or dislike certain part of a road), which are not always practical or easy to obtain and/or estimate (on a daily basis). Therefore, a route planning system offering a set of good/reasonable alternatives can hope that (at least) one of them can satisfy the user, and vice versa, the user can have them as back-up choices for altering his route in case of emergent traffic conditions. In all cases, the essential task is to compute reasonable alternatives to an $s$-$t$ optimal route and this has to be done fast.

In this context, we are witnessing some recent research which investigates the computation of alternative routes under two approaches. The first approach, initiated in [5] and further extended in [19, 16], computes a few (2-3) alternative $s$-$t$ routes that pass through specific nodes (called *via nodes*). The second approach [6] creates a set of reasonable alternative routes in the form of a graph, called *alternative graph*. Moreover, there are proprietary algorithms used by commercial systems (e.g., by Google and TomTom) that suggest alternative routes.

* This work was supported by the EU FP7/2007-2013 (DG CONNECT.H5-Smart Cities & Sustainability), under grant agreement no. 288094 (project eCOMPASS).

In this work, we focus on computing alternative graphs, which appear to be more suitable for practical navigation systems [4, 18], since the approach with via-nodes may create higher (than required) overlapping and may not be always successful. The study in [6] quantified the quality characteristics of an alternative graph (AG), captured by three criteria. These concern the non-overlappingness and the stretch of the routes, as well as the number of decision edges (sum of node out-degrees) in AG. As it is shown in [6], all of them together are important in order to produce a high-quality AG. However, optimizing a simple objective function combining just any two of them is already an NP-hard problem [6]. Hence, one has to concentrate on heuristics. Four heuristic approaches were investigated in [6] with those based on Plateau [3], Penalty [7], and a combination of them to be the best.

In this paper, we extend the approach in [6] for building AGs in two directions. First, we introduce a pruning stage that precedes the execution (and it is independent) of any heuristic method, thus reducing the search space and hence detecting the nodes on shortest routes much faster. Second, we provide several improvements on both the Plateau and Penalty methods. In particular, we use a different approach for filtering plateaus in order to identify the best plateaus that will eventually produce the most qualitative alternative routes, in terms of minimum overlapping and stretch. We also introduce a practical and well-performed combination of the Plateau and Penalty methods with tighter lower-bounding based heuristics. This has the additional advantage that the lower bounds remain valid for use even when the edge costs are increased (without requiring new preprocessing), and hence are useful in dynamic environments where the travel time may be increased, for instance, due to traffic jams. Finally, we conducted an experimental study for verifying our methods on several road networks of Western Europe. Our experiments showed that our methods can produce AGs of high quality pretty fast.

This paper is organized as follows. Section 2 provides background information, including notation, formal problem definitions, and classic algorithms for the *single pair shortest path problem*. Section 3 surveys the methods for building alternative graphs, presented in [6]. Section 4 presents our proposed improvements for producing AGs of better quality. Section 5 presents a thorough experimental evaluation of our improved methods. Conclusions are offered in Section 6.

**Recent related work.** During preparation of our camera-ready version, we have been informed about a different approach on reducing the running time of the Penalty method [17], which is is based on Customizable Route Planning [8] and includes an iterative updating of shortest path heuristics through a multi-level partition, in order to accommodate the adjustments on the edge weights by the Penalty method.

## 2 Preliminaries

A *road network* can be modeled as a *directed graph* $G = (V, E)$, where each node $v \in V$ represents intersection points along roads, and each edge $e \in E$ represents road segments between pairs of nodes. Let $|V| = n$ and $|E| = m$.

We consider the problem of tracing alternative paths from a source node $s$ to a target node $t$ in $G$, with edge weight or cost function $w : E \to \mathbb{R}^+$. The essential goal is to obtain sufficiently different paths with optimal or near optimal cost.

### 2.1 Alternative Graphs

The aggregation of alternative paths between a source $s$ and a target $t$ can be captured by the concept of the *Alternative Graph*, a notion first introduced in [6]. An Alternative Graph

(AG) is defined as the union of several $s$-$t$ paths. Formally, an $AG$ $H = (V', E')$ is a graph, with $V' \subseteq V$, and such that $\forall e = (u, v) \in E'$, there is a $P_{uv}$ path in $G$ and a $P_{st}$ path in $H$, so that $e \in P_{st}$ and $w(e) = w(P_{uv})$, where $w(P_{uv})$ denotes the weight or cost of path $P_{uv}$. Let $d(u, v) \equiv d_G(u, v)$ be the shortest distance from $u$ to $v$ in graph $G$, and $d_H(u, v)$ be the shortest distance from $u$ to $v$ in graph $H$.

Storing paths in an alternative graph $AG$ makes sense, because in general alternative paths may share common nodes (including $s$ and $t$) and edges. Furthermore, their subpaths may be combined to form new alternative paths.

In the general case, there may be several alternative paths from $s$ to $t$. Hence, there is a need of filtering and rating all alternatives, based on certain quality criteria. The main idea of the quality criteria is to discard routes with poor rates. For this task, the following quality indicators were used in [6]:

$$\begin{aligned} \mathit{totalDistance} &= \sum_{e=(u,v)\in E'} \frac{w(e)}{d_H(s,u) + w(e) + d_H(v,t)} && (\mathit{overlapping}) \\ \mathit{averageDistance} &= \frac{\sum_{e\in E'} w(e)}{d_G(s,t) \cdot \mathit{totalDistance}} && (\mathit{stretch}) \\ \mathit{decisionEdges} &= \sum_{v\in V'\setminus\{t\}} (\mathit{outdegree}(v) - 1) && (\mathit{size\ of\ AG}) \end{aligned}$$

In the above definitions, the *totalDistance* measures the extend to which the paths in $AG$ are non-overlapping. Its maximum value is *decisionEdges*+1. This is equal to the number of all $s$-$t$ paths in $AG$, when these are disjoint, i.e. not sharing common edges. The *averageDistance* measures the average cost of the alternatives compared with the shortest one (i.e. the stretch). Its minimum value is 1. This occurs, when every $s$-$t$ path in $AG$ has the minimum cost. Consequently, to compute a qualitative $AG$, one aims at high *totalDistance* and low *averageDistance*. The *decisionEdges* measures the size complexity of $AG$. In particular, the number of the alternative paths in $AG$, depend on the "decision branches" are in $AG$. For this reason, the higher the *decisionEdges*, the more confusion is created to a typical user, when he tries to decide his route. Therefore, it should be bounded.

## 2.2 Shortest path Heuristics

We review now some shortest path heuristics that will be used throughout the paper.

**Forward Dijkstra.** Recall that Dijkstra's algorithm [11] grows a full shortest path tree rooted at a source node $s$, by performing a breadth-first based search, exploring the nodes in $G$ in increasing order of distance from $s$. More specifically, for every node $u$, the algorithm maintains a tentative distance from $s$ of the current known $s$-$u$ shortest path and the predecessor node *pred* of $u$ on this path. The exploring and processing order of the nodes can be controlled and guided by a priority queue $Q$. In each iteration, the node $u$ with the minimum tentative distance $d(s, u)$ is removed from $Q$ and its outgoing edges are *relaxed*. More specifically, for any outgoing edge of $u$, $e = (u, v) \in E$, if $d(s, u) + w(e) < d(s, v)$ then it sets $d(s, v) = d(s, u) + w(e)$ and $\mathit{pred}(v) = u$. Because the distance from $s$ is monotonically increasing ($w : E \to \mathbb{R}^+$) a node dequeued from $Q$ becomes *settled*, receiving the minimum possible distance $d(s, v)$ from $s$. The algorithm terminates when the queue becomes empty or when a target $t$ is settled (for single-pair shortest path queries). In the latter case the produced shortest path tree consists of nodes with $d(s, v) \leqslant d(s, t)$. We refer to Dijkstra's algorithm also as *forward Dijkstra*.

**Backward Dijkstra.** To discover the shortest paths from all nodes in $G$ to a target node $t$ we can use a backward version of Dijkstra's algorithm. The *backward Dijkstra* explores the nodes in $G$ in increasing order of their distance to $t$, traversing the incoming edges of the nodes by the reverse direction. In this variant of Dijkstra's algorithm, the successors ($succ$) nodes are stored instead of the predecessor ones in order to orientate the direction of the built shortest path tree towards the target node $t$.

The following bidirectional and A* variants of Dijkstra's algorithm are used to reduce the expensive and worthless exploration on nodes that do not belong to a shortest $s$-$t$ path.

**Bidirectional Dijkstra.** This bidirectional variant runs forward Dijkstra from $s$ and backward Dijkstra from $t$, as two simultaneously auxiliary searches. Specifically, the algorithm alternates the forward search from $s$ and the backward (reverse) search from $t$, until they meet each other. In this way, the full $s$-$t$ shortest path is formed by combining a $s$-$v$ shortest path computed by the forward search and a $v$-$t$ shortest path computed by the backward search. Because two $s$-$v$ and $v$-$t$ shortest paths cannot necessarily build an entire shortest $s$-$t$ path, additionally, there is a need to keep the minimum cost $w(s, v) + w(v, t)$ and the via node $v$ from all current traced paths in the meeting points of the two searches. Let $d_s(u) = d(s, u)$ and $d_t(u) = d(u, t)$. The algorithm terminates after acquiring the correct $s$-$t$ shortest path. This is ensured only when the current minimum distance in the priority queue of forward search $Q_f$ and the minimum distance in the priority queue $Q_b$ of backward search are such that $\min_{u\in Q_f}\{d_s(u)\} + \min_{v\in Q_b}\{d_t(v)\} > d_s(t)$, meaning that the algorithm cannot anymore provide shorter $s$-$t$ paths than the previous discovered ones.

**A* search.** Given a source node $s$ and a target node $t$, the $A^*$ variant [15] is similar to Dijkstra's algorithm with the difference that the priority of a node in $Q$ is modified according to a heuristic function $h_t : V \to \mathbb{R}$ which gives a lower bound estimate $h_t(u)$ for the cost of a path from a node $u$ to a target node $t$. By adding this heuristic function to the priority of each node, the search becomes goal-directed pulling faster towards the target. The tighter the lower bound is, the faster the target is reached. The only requirement is that the $h_t$ function must be monotone: $h_t(u) \leq w(u, v) + h_t(v)$, $\forall (u, v) \in E$. One such heuristic function is the Euclidean distance between two nodes. But in general, Euclidean lower bounds are not the best approximations of shortest distances in road networks, because the majority of road routes do not follow a strict straight course from $s$ to $t$.

**ALT.** The *ALT* technique, that introduced in [13], provides a highly effective heuristic function for the $A^*$ algorithm, using triangle inequality and precomputed shortest path distances between all nodes and few important nodes, the so-called *landmarks*. Those shortest distances can be computed and stored in a preprocessing stage. Then, during a query, the lower bounds can be estimated in constant time. In particular, for a node $v$ and a landmark $L$, it holds that $d(v, t) \geqslant \max_L\{d(L, t) - d(L, v), d(v, L) - d(t, L)\} = h_t(v)$ and $d(s, v) \geqslant max_L\{d(L, v) - d(L, s), d(s, L) - d(v, L)\} = h_s(v)$. Obviously, these lower bounds contain an important part of the information of the shortest path trees in $G$.

The efficiency of ALT depends on the number and the initial selection of landmarks. In order to have good query times, peripheral landmarks as far away from each other as possible must be selected, taking advantage of the fact that the road networks are (almost) planar. The nodes in these positions can cover more shortest path trees in $G$ and hence provide more valuable heuristics.

We refer to the consistent bidirectional *ALT* algorithm, with the average heuristic function [13], as *BLA*. In this variant, the forward and backward search use $H_s$ and $H_t$ as heuristic functions, where $H_t(v) = -H_s(v) = \frac{h_t(v)-h_s(v)}{2}$.

## 3 Approaches for Computing Alternative Graphs

We briefly review the approaches considered in [6] for computing alternative graphs.

**k-Shortest Paths.** The $k$-shortest path routing algorithm [12, 22] finds $k$ shortest paths in order of increasing cost. The disadvantage of this approach is that the computed alternative paths share many edges, which makes them difficult to be distinguished by humans. Good alternatives could be revealed for very large values of $k$, but at the expense of a rather high computational cost.

**Pareto.** The Pareto algorithm [14, 21, 10] computes an $AG$ by iteratively finding *Pareto-optimal* paths on a suitably defined objective cost vector. The idea is to use as first edge cost the one of the single criterion problem, while the second edge cost is defined as follows: all edges belonging to $AG$ (initially the $AG$ is the shortest $s$-$t$ path) set their second cost function to their initial edge cost and all edges not belonging to $AG$ set their second cost function to zero.

**Plateau.** The Plateau method [3] provides alternative $P_{st}$ paths by connecting pairs of $s$-$v$ and $v$-$t$ shortest paths, via a specific node $v$. In this matter, $v$ is selected on the basis of whether it belongs to a plateau (to be defined shortly).

In particular, the $s$-$v$ and $v$-$t$ paths that are required to form the $P_{st}$ paths can be found on a forward $T_f$ shortest path tree, with root $s$, and a backward $T_b$ shortest path tree, with root $t$. On this, a classical approach for finding $T_f$ and $T_b$ is by performing forward and backward Dijkstra. Apparently, from this process, connecting shortest subpaths does not necessarily ensure the optimality of the resulted $P_{st}$ paths, so there is a need to evaluate them. In order to provide low overlapping alternative $P_{st}$ paths, the connection-node $v$ of $s$-$v$ and $v$-$t$ paths should belong to a plateau. The plateaus are simple paths, $\overline{P} \subseteq P_{st}$, consisting of more than one successive nodes, with the property that $\forall u, v \in \overline{P} : d_s(u) + d_t(u) = d_s(v) + d_t(v)$. The plateaus can be traced on the intersection of $T_f$ and $T_b$. In this way, a node in a plateau following the predecessor nodes in $T_f$ and the successor nodes in $T_b$ can build a complete

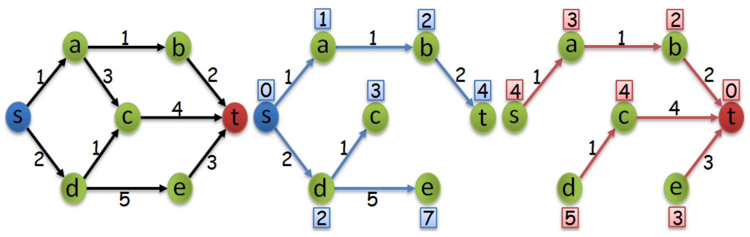


**Figure 1** Graph $G$. Forward $T_f$ shortest path tree with root $s$. Backward $T_b$ shortest path tree with root $t$.

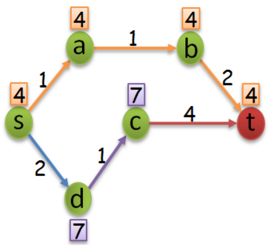


**Figure 2** The combination of $T_f$ and $T_b$ trees. The resulted graph reveals two plateaus. The first one is $s$-$a$-$b$-$t$ and the second one is $d$-$c$.

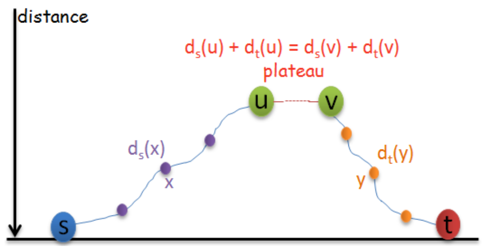


**Figure 3** A Plateau.

$P_{st}$ path. As the plateaus are usually too many, a filtering stage is used to select the best of them. In [6], this is implemented by gathering plateaus $\overline{P}$ in a non-decreasing order of $rank = w(P_{st}) - w(\overline{P})$, where $P_{st}$ is the resulted path via $\overline{P}$. Therefore, a plateau that corresponds to a shortest path from $s$ to $t$ has rank zero, which is the best value.

**Penalty.** The Penalty method [7] provides alternative paths by iteratively running shortest path queries and adjusting the weight of the edges on the resulted $P_{st}$ paths. The basic steps are the following. A shortest $P_{st}$ path is computed with Dijkstra's algorithm or a speedup variation of it. Then, $P_{st}$ is penalized by increasing the weight of its edges. Next, a new $s$-$t$ query is executed. If the new computed $P'_{st}$ path is short and different enough from the previously discovered $P_{st}$ paths, then it is added to the solution set. The same process is repeated until a sufficient number of alternative paths (with the desired characteristics) is found, or the weight adjustments of $s$-$t$ paths bring no better results.

In order to offer the best results, an efficient and safe way on weight increases should be adopted. A weight adjustment policy, also considered in [6], is as follows:

- The increase on the weights should be of a small magnitude in order to keep the resulted *averageDistance* low. When an edge of a $P_{st}$ path is about to be penalized, only a small fraction (penalty factor) $0.1 \leq p \leq 1$ of its initial weight is added, i.e., $w_{new}(e) = w(e) + p \cdot w_{old}(e)$. Note that the use of constant values is avoided, because they do not always guarantee a balanced adjustment, since in some cases longer edges may be favored over shortest ones. In general, the higher the penalty factor is, the more the new shortest path may differ from the last one. On the other hand, the lower the $p$ penalty factor is, more shortest path queries can be performed and less alternative paths may be lost.
- The weight adjustment is restricted when it could lead to the loss of good alternatives. Notice that, an unbounded penalty leads to multiple increases on the edge weights and is risky. For example, suppose that there is only one fast highway into a city, whereas there are many alternatives through the city center. If we allow multiple increases on the weights of the highway then its cost will be increased several times during the iterations. This for new $s$-$t$ shortest path queries may result to new computed paths that now begin from a detour longer than the highway. Therefore, because of the high cost any possible alternative inside the city will be lost, and the algorithm will terminate with poor results. To overcome this problem the number of the increases or the magnitude of $p$ is limited for edges already included in $AG$.
- In order to avoid the overlapping between the computed alternative paths is useful to extend the weight adjustment to their neighborhood. This is reasonable, because in some cases the new computed alternative paths may share many small detours with the previous ones. For example, it is possible that the first path is a fast highway and the new computed paths are in the same course with the highway but having one or

many outgoing and incoming small detours distributed along the highway. This increases the *decisionEdges* and offers meaningless (non-discrete) alternatives. Therefore, when increasing the weight of the edges in a shortest $P_{st}$ path, the weights of edges around $P_{st}$ that leave and join the current $AG$ should be additionally penalized (rejoin-penalty) by a factor $0.1 \leq r \leq 1$. Consequently, the rejoin-penalty $r$ contributes to high *totalDistance*.

**Thinout.** A major issue is the optimality regarding the cost. In [6], the optimality is ensured by bounding the *averageDistance*, and further in a post-processing phase by setting tighter bounds to the local optimality of the edges or the subpaths in $AG$. In Plateau method, the local optimality of the $s$-$v$ and $v$-$t$ paths is guaranteed because these are selected from the $T_f$ and $T_b$ shortest path trees. In the penalty method, however, the adjustment of the weights may insert non optimal paths. A way in [6] to overcome this issue, when considering alternative paths globally, is by performing a *global refinement* (focusing on the entire $s$-$t$ paths), and an iterative *local refinement* (focusing on individual edges). In more detail, for some $\delta > 1$, in global refinement, an edge $e = (u, v) \in E'$ is removed from $AG$ if $d_H(s, u) + w(u, v) + d_H(v, t) > \delta \cdot d_H(s, t)$. In local refinement, an edge $e = (u, v) \in E'$ is removed from $AG$ if $w(u, v) > \delta \cdot d_H(u, v)$.

## 4 Our improvements

As the experimental study in [6] showed, the $k$-Shortest Paths and the Pareto approaches generate alternative graphs of low quality and hence we shall not investigate them. On the other hand, the Plateau and Penalty methods are the most promising ones and thus we focus on extending and enhancing them. Our improvements are twofold :

**A.** We introduce a pruning stage that precedes the Plateau and Penalty methods in order to a-priori reduce their search space without sacrificing the quality of the resulted alternative graphs.

**B.** We use a different approach for filtering plateaus in order to obtain the ones that generate the best alternative paths. In addition, we fine tune the penalty method, by carefully choosing the penalizing factors on the so far computed $P_{st}$ paths, in order to trace the next best alternatives.

### 4.1 Pruning

We present two bidirectional Dijkstra-based *pruners*. The purpose of both of them, is to identify the nodes that are in $P_{st}$ shortest paths. We refer to such nodes, as the useful search space, and the rest ones, as the useless search space. Our goal, through the use of search pruners, is to ensure: (a) a more quality-oriented $AG$ construction and (b) a reduced dependency of the time computation complexity from the graph size. The latter is necessary, in order to acquire fast response in queries. We note that the benefits are notably for the Penalty method. This is because, the Penalty method needs to run iteratively several $s$-$t$ shortest path queries. Thus, having put aside the useless nodes and focussing only on the useful ones, we can get faster processing. We also note that, over the $P_{st}$ paths with the minimum cost, may be desired as well to let in $AG$ paths with near optimal cost, say $\tau \cdot d_s(t)$, which will be the maximum acceptable cost $w(P_{st})$. Indicatively, $1 \leqslant \tau \leqslant 1.4$. Obviously, nodes far away from both $s$ and $t$, with $d_s(v) + d_t(v) > \tau \cdot d_s(t)$, belong to $P_{st}$ paths with prohibitively high cost. In the following we provide the detailed description of both pruners, which is illustrated by Figures 4 and 5.

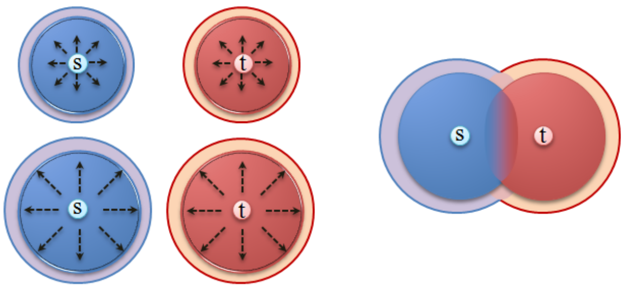


**Figure 4** The forward and backward searches meet each other. In this phase the minimum distance $d_s(t)$ is traced.

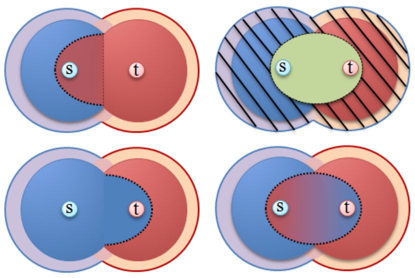


**Figure 5** The forward and backward settles only the nodes in the shortest paths, taking account the overall $d_s(v) + d_t(v)$.

**Uninformed Bidirectional Pruner.** In this pruner, there is no preprocessing stage. Instead, the used heuristics are obtained from the minimum distances of the nodes enqueued in $Q_f$ and $Q_b$, i.e. $Q_f.minKey() = \min_{u \in Q_f}\{d_s(u)\}$ and $Q_b.minKey() = \min_{v \in Q_b}\{d_t(v)\}$.

We extend the regular bidirectional Dijkstra, by adding one extra phase. First, for computing the minimum distance $d_s(t)$, we let the expansion of forward and backward search until $Q_f.minKey() + Q_b.minKey() \geq d_s(t)$. At this step, the current forward $T_f$ and backward $T_b$ shortest path trees produced by the bidirectional algorithm will have crossed each other and so the minimum distance $d_s(t)$ will be determined. Second, at the new extra phase, we continue the expansion of $T_f$ and $T_b$ in order to include the remaining useful nodes, such that $d_s(v) + d_t(v) \leq \tau \cdot d_s(t)$, but with a different mode. This time, we do not allow the two searches to continue their exploration at nodes $v$ that have $d_s(v) + h_t(v)$ or $h_s(v) + d_t(v)$ greater than $\tau \cdot d_s(t)$. We use the fact that $Q_f$ and $Q_b$ can provide lower-bound estimates for $h_s(v)$ and $h_t(v)$. Specifically, a node that is not settled or explored from backward search has as a lower bound to its distance to $t$, $h_t(v) = Q_b.minKey()$. This is because the backward search settles the nodes in increasing order of their distance to $t$, and if $u$ has not been settled then it must have $d_t(u) \geq Q_b.minKey()$. Similarly, a node that is not settled or explored from forward search has a lower bound $h_s(v) = Q_f.minKey()$. Furthermore, when a search settles a node that is also settled from the other search we can calculate exactly the sum $d_s(u) + d_t(u)$. In this case, the higher the expansion of forward and backward search is, the more tight the lower bounds become. The pruning is ended, when $Q_f$ and $Q_b$ are empty. Before the termination, we exclude the remaining useless nodes that both searches settled during the pruning, that is all nodes $v$ with $d_s(v) + d_t(v) > \tau \cdot d_s(t)$.

**Informed ALT bidirectional pruner.** In the second pruner, our steps are similar, except that we use tighter lower bounds. We acquire them in an one-time preprocessing stage, using the *ALT* approach. In this case, the lower bounds that are yielded can guide faster and more accurately the pruning of the search space. We compute the shortest distances between the nodes in $G$ and a small set of landmarks. For tracing the minimum distance $d_s(t)$, we use *BLA* as base algorithm, which achieves the lowest waste exploration, as experimental results showed in [13, 20]. During the pruning, we skip the nodes that have $d_s(v) + h_t(v)$ or $h_s(v) + d_t(v)$ greater than $\tau \cdot d_s(t)$.

The use of lower-bounding heuristics can be advantageous. In general, a heuristic stops being valid when a change in the weight of the edges is occurred. But note that in the penalty method, we consider only increases on the edge weights and therefore this does not affect the lower bounds on the shortest distances. Therefore, the combination of the *ALT* speedup [20, 13] with Penalty is suitable. However, depending on the number and the magnitude of the increases the lower bounds can become less tight for the new shortest distances, leading to a reduced performance on computing the shortest paths.

## 4.2 Filtering and Fine-tuning

Over the standard processing operations of Penalty and Plateau, we introduce new ones for obtaining better results. In particular:

**Plateau.** We use a different approach on filtering plateaus. Specifically, over the cost of a plateau path we take account also its non-overlapping with others. In this case, the difficulty is that the candidate paths may share common edges or subpaths, so the *totalDistance* is not fixed. Since at each step an insertion of the current best alternative path in $AG$ may lead to a reduced *totalDistance* for the rest candidate alternatives, primarily we focus only on their unoccupied parts, i.e., those that are not in $AG$. We rank a $x$-$y$ plateau $\overline{P}$ with $rank = totalDistance - averageDistance$, where $totalDistance = \frac{w(\overline{P})}{d_s(x)+w(\overline{P})+d_t(y)}$ is its definite non-overlapping degree, and $averageDistance = \frac{w(\overline{P})+d_s(t)}{(1+totalDistance)\cdot d_s(t)}$ is its stretch over the shortest $s$-$t$ path in $G$. During the collection of plateaus, we insert the highest in rank of them via its node-connectors $v \in \overline{P}$ in $T_f$ and $T_b$ to a min heap with fixed size equal to *decisionEdges* plus an offset. The offset increases the number of the candidate plateaus, when there are available, and it is required only as a way out, in the case, where several $P_{st}$ paths via the occupied plateaus in $AG$ lead to low *totalDistance* for the rest $P_{st}$ paths via the unoccupied plateaus.

**Penalty.** When we "penalize" the last computed $P_{st}$ path, we adjust the increases on the weights of its outgoing and incoming edges, as follows:

$$w_{new}(e) = w(e) + (0.1 + r \cdot d_s(u)/d_s(t)) \cdot w_{old}(e), \quad \forall e = (u,v) \in E: \ u \in P_{st},\ v \notin P_{st}$$
$$w_{new}(e) = w(e) + (0.1 + r \cdot d_t(v)/d_t(s)) \cdot w_{old}(e), \quad \forall e = (u,v) \in E: \ u \notin P_{st},\ v \in P_{st}$$

The first adjustment puts heavier weights on those outgoing edges that are closer to the target $t$. The second adjustment puts heavier weights on those incoming edges that are closer to the source $s$. The purpose of both is to reduce the possibility of recomputing alternative paths that tend to rejoin directly with the previous one traced.

An additional care is given also for the nodes $u$ in $P_{st}$, having $outdegree(u) > 1$. Note that their outgoing edges can form different branches. Since the edge-branches in $G$ constitute generators for alternative paths, they are important. These edges are being inserted to $AG$ with a greater magnitude of weight increase than the rest of the edges.

The insertion of the discovered alternative paths in $G$ and the maintenance of the overall quality of $AG$ should be controlled online. Therefore, we establish an online interaction with the $AG$'s quality indicators, described in Section 2, for both Plateau and Penalty. This is also necessary because, at each step an insertion of the current best alternative may lead to a reduced value of *totalDistance* for the next candidate alternative paths that share common edges with the already computed $AG$.

In order to get the best alternatives, we seek to maximize the $targetfunction = totalDistance - \alpha \cdot averageDistance$, where $\alpha$ is a balance factor that adjusts the stretch magnitude rather than the overlapping magnitude. Maximization of the target function leads to select the best set of low overlapping and shortest alternative paths.

Since the penalty method can work on any pre-computed $AG$, it can be combined with Plateau. In this way, we collect the best alternatives from Penalty and Plateau, so that the resulting set of alternatives maximizes the target function. In this matter, we can extend the number of decision edges and after the gathering of all alternatives, we end by performing thinout in $AG$. Moreover, in order to guide the Penalty method to the remaining alternatives, we set a penalty on the paths stored by Plateau in $AG$, by increasing their weights. We also use the same pruning stage to accommodate both of them.

## 5 Experimental Results

The experiments were conducted on an Inte(R) Xeon(R) Processor X3430 @ 2.40GHz, with a cache size of 8Mb and 32Gb of RAM. Our implementations were written in C++ and compiled by GCC version 4.6.3 with optimization level 3.

The data sets of the road networks in our experiments were acquired from OSM [1] and TomTom [2]. The weight function is the travel time along the edges. In the case of OSM, for each edge, we calculated the travel time based on the length and category of the roads (residential street, tertiary, secondary, primary road, trunk, motorway, etc). The data set of the Greater Berlin area was kindly provided by TomTom in the frame of the eCOMPASS project [4]. The size of the data sets are reported in Table 1.

For our implementations, we used the packed-memory graph (PMG) structure [20]. This is a highly optimized graph structure, part of a larger algorithmic framework, specifically suited for very large scale networks. It provides dynamic memory management of the graph and thus the ability to control the storing scheme of nodes and edges in memory for optimization purposes. It supports almost optimal scanning of consecutive nodes and edges and can incorporate dynamic changes in the graph layout in a matter of $\mu s$. The ordering of the

**Table 1** The size of road networks.

| map | | $n$ | $m$ |
|---|---|---|---|
| B | Berlin | 117,839 | 310,152 |
| LU | Luxembourg | 51,576 | 119,711 |
| BE | Belgium | 576,465 | 1,376,142 |
| IT | Italy | 2,425,667 | 5,551,700 |
| GB | GreatBritain | 3,233,096 | 7,151,300 |
| FR | France | 4,773,488 | 11,269,569 |
| GE | Germany | 7,782,773 | 18,983,043 |
| WE | WesternEurope | 26,498,732 | 62,348,328 |

nodes and edges in memory is in such a way that increases the locality of references, causing as few memory misses as possible and thus a reduced running time for the used algorithms.

We tested our implementations in the road network of the Greater Berlin area, the Western Europe (Austria, Belgium, Denmark, France, Germany, Italy, Luxembourg, Netherlands, Norway, Portugal, Spain, Sweden, Switzerland, and Great Britain), as well as in the network of each individual West European country. In the experiments, we considered 100 queries, where the source $s$ and the destination $t$ were selected uniformly at random among all nodes. For the case of the entire Western Europe's road network, the only limitation is that the $s$-$t$ queries are selected, such that their geographical distance is at most 300 kilometers. This was due to the fact that although modern car navigation systems may store the entire maps, they are mostly used for distances up to a few hundred kilometers.

For far apart source and destination, the search space of the alternative $P_{st}$ paths gets too large. In such cases, it is more likely that many non-overlapping long (in number of edges) paths exist between $s$ and $t$. Therefore, this has a major effect on the computation cost of the overall alternative route planning. In general, the number of non-overlapping shortest paths depends on the density of the road networks as well on the edge weights.

There is a trade-off between the quality of $AG$ and the computation cost. Thus, we can sacrifice a bit of the overall quality to reduce the running time. Consequently, in order to deal with the high computation cost of the alternative route planning for far apart sources and destinations we can decrease the parameter $\tau$ (max stretch). A dynamic and online adjustment of $\tau$ based on the geographical distance between source and target can be used too. For instance, at distance larger than 200km, we can set a smaller value to $\tau$, e.g., close to 1, to reduce the stretch and thereby the number of the alternatives. We adopted this arrangement on large networks (Germany, Western Europe). In the rest, we set $\tau = 1.2$, which means that any traced path has cost at most 20% larger than the minimum one. To all road networks, we also set $averageDistance \leq 1.1$ to ensure that, in the filtering stage, the average cost of the collected paths is at most 10% larger than the minimum one.

In order to fulfill the ordinary human requirements and deliver an easily representable $AG$, we have bounded the *decisionEdges* to 10. In this way, the resulted $AG$ has small size, $|V'| \ll |V|$ and $|E'| \ll |E|$, thus making it easy to store or process. Our experiments showed that the size of an $AG$ is at most 2 to 3 times the size of a shortest $s$-$t$ path, which we consider as a rather acceptable solution.

Our base *target function* [1] in Plateau and Penalty is $totalDistance - averageDistance + 1$. Regarding the pruning stage of Plateau and Penalty, we have used the ALT-based informed bidirectional pruner with at most 24 landmarks for Western Europe.

In Tables 2, 3, and 4, we report the results of our experiments on the various quality indicators: targetFunction (*TargFun*), totalDistance (*TotDist*), averageDistance (*AvgDist*) and decisionEdges (*DecEdges*). The values in parentheses in the header columns provide only the theoretically maximum or minimum values per quality indicator, which may be far away from the optimal values (that are based on the road network and the $s$-$t$ queries).

In Tables 2, 3, and 4, we report the average value per indicator. The overall execution time for computing the entire $AG$ is given in milliseconds. As we see, we can achieve a high-quality $AG$ in less than a second even for continental size networks. The produced alternative paths in $AG$ are directly-accessible for use (e.g., they are not stored in any compressed form).

[1] We have been very recently informed [9] that this is the same target function as the one used in [6] and not the erroneously stated $totalDistance - averageDistance$ in that paper.

**Table 2** The average quality of the resulted *AG* via Plateau method.

| map | TargFun | | TotDist | AvgDist | DecEdges | Time |
|---|---|---|---|---|---|---|
| | (max:11) | in [6] | (max:11) | (min:1) | (max:10) | (ms) |
| B | 3.82 | - | 3.91 | 1.09 | 9.95 | 45.61 |
| LU | 4.44 | 3.05 | 4.49 | 1.05 | 9.73 | 37.05 |
| BE | 4.83 | - | 4.87 | 1.04 | 10.00 | 85.08 |
| IT | 4.10 | - | 4.14 | 1.04 | 9.92 | 114.29 |
| GB | 4.36 | - | 4.40 | 1.04 | 9.93 | 180.12 |
| FR | 4.22 | - | 4.26 | 1.04 | 9.97 | 159.93 |
| GE | 4.88 | - | 4.92 | 1.04 | 10.00 | 286.40 |
| WE | 4.35 | 3.08 | 4.37 | 1.02 | 9.88 | 717.57 |

**Table 3** The average quality of the resulted *AG* via Penalty method.

| map | TargFun | | TotDist | AvgDist | DecEdges | Time |
|---|---|---|---|---|---|---|
| | (max:11) | in [6] | (max:11) | (min:1) | (max:10) | (ms) |
| B | 4.16 | - | 4.23 | 1.07 | 9.92 | 49.34 |
| LU | 5.14 | 2.91 | 5.19 | 1.05 | 9.23 | 41.56 |
| BE | 5.29 | - | 5.33 | 1.04 | 9.54 | 159.71 |
| IT | 4.11 | - | 4.14 | 1.03 | 9.47 | 105.84 |
| GB | 4.38 | - | 4.41 | 1.03 | 9.87 | 210.94 |
| FR | 4.11 | - | 4.16 | 1.05 | 9.32 | 192.44 |
| GE | 5.42 | - | 5.46 | 1.04 | 9.91 | 388.97 |
| WE | 5.21 | 3.34 | 5.24 | 1.03 | 9.67 | 776.97 |

**Table 4** The average quality of the resulted *AG* via the combined Penalty and Plateau method.

| map | TargFun | | TotDist | AvgDist | DecEdges | Time |
|---|---|---|---|---|---|---|
| | (max:11) | in [6] | (max:11) | (min:1) | (max:10) | (ms) |
| B | 4.55 | - | 4.61 | 1.06 | 9.97 | 54.12 |
| LU | 5.25 | 3.29 | 5.30 | 1.05 | 9.81 | 43.69 |
| BE | 5.36 | - | 5.41 | 1.05 | 9.89 | 163.75 |
| IT | 4.37 | - | 4.41 | 1.04 | 9.79 | 178.08 |
| GB | 4.67 | - | 4.71 | 1.04 | 9.86 | 284.38 |
| FR | 4.56 | - | 4.60 | 1.04 | 9.86 | 217.30 |
| GE | 5.50 | - | 5.54 | 1.04 | 9.89 | 446.38 |
| WE | 5.49 | 3.70 | 5.52 | 1.03 | 9.94 | 987.42 |

**Table 5** Alternative route queries in the road network of Western Europe, with geographical distance up to 500km and $\tau$ value of up to 1.2.

| map WE | TargFun | TotDist | AvgDist | DecEdges |
|---|---|---|---|---|
| Plateau | 4.71 | 4.73 | 1.02 | 10.00 |
| Penalty | 6.46 | 6.48 | 1.02 | 9.97 |
| Plateau & Penalty | 6.82 | 6.84 | 1.02 | 9.98 |

Due to the limitation on the number of the decision edges in *AG* and the low upper bound in stretch, we have chosen in the Penalty method small penalty factors, $p = 0.1$ and $r = 0.1$. In addition, this serves in getting better low-stretch results, see Table 3. In contrast, the *averageDistance* in Plateau gets slightly closer to the 1.1 upper bound.

In our experiments, the Penalty method clearly outperforms Plateau on finding more qualitative results. However it has higher computation cost. This is reasonable because it needs to perform around to 10 shortest $s$-$t$ path queries. The combination of Penalty and Plateau is used to extract the best results of both of the methods. Therefore in this way the resulted *AG* has better quality than the one provided by any individual method. In Tables 2, 3, and 4, we also report on the *TargFun* quality indicator of the study in [6]. The experiments in that study were run only on the LU and WE networks, and on data provided by PTV, which concerned smaller in size networks and which may be somehow different from those we use here [1]. Nevertheless, we put the *TargFun* values in [6] as a kind of reference for comparison.

We would like to note that if we allow a larger value of $\tau$ (up to 1.2) for large networks (e.g., WE) and for $s$-$t$ distances larger than 300km, then we can achieve higher quality indicators (intuitively, this happens due to the much more alternatives in such a case). Indicative values of quality indicators for WE are reported in Table 5.

## 6 Conclusion

We have extended the Penalty and Plateau based methods in [6] as well as their combination in several ways. We can generate a large number of qualitative alternatives with high non-overlappingness and low stretch in time less than 1 second on continental size networks. The new heuristics can tolerate edge cost increases without requiring new preprocessing.

## References

1 Openstreetmap. http://www.openstreetmap.org.

2 Tomtom. http://www.tomtom.com.

3 Camvit: Choice routing, 2009. http://www.camvit.com.

4 eCOMPASS project, 2011-2014. http://www.ecompass-project.eu.

5 Ittai Abraham, Daniel Delling, Andrew V Goldberg, and Renato F Werneck. Alternative routes in road networks. In *Experimental Algorithms (SEA)*, pages 23–34. Springer, 2010.

6 Roland Bader, Jonathan Dees, Robert Geisberger, and Peter Sanders. Alternative route graphs in road networks. In *Theory and Practice of Algorithms in (Computer) Systems*, pages 21–32. Springer, 2011.

7 Yanyan Chen, Michael GH Bell, and Klaus Bogenberger. Reliable pretrip multipath planning and dynamic adaptation for a centralized road navigation system. *Intelligent Transportation Systems, IEEE Transactions on*, 8(1):14–20, 2007.

8 Daniel Delling, Andrew V Goldberg, Thomas Pajor, and Renato F Werneck. Customizable route planning. In *Experimental Algorithms*, pages 376–387. Springer, 2011.

9 Daniel Delling and Moritz Kobitzsch. Personal commnication, July 2013.

10 Daniel Delling and Dorothea Wagner. Pareto paths with SHARC. In *Experimental Algorithms*, pages 125–136. Springer, 2009.

11 Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

12 David Eppstein. Finding the k shortest paths. *SIAM Journal on computing*, 28(2):652–673, 1998.

13 Andrew V Goldberg and Chris Harrelson. Computing the shortest path: A* search meets graph theory. In *Proc. 16th ACM-SIAM symposium on Discrete algorithms*, pages 156–165. Society for Industrial and Applied Mathematics, 2005.

14 Pierre Hansen. Bicriterion path problems. In *Multiple criteria decision making theory and application*, pages 109–127. Springer, 1980.

15 Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *Systems Science and Cybernetics, IEEE Transactions on*, 4(2):100–107, 1968.

16 Moritz Kobitzsch. An alternative approach to alternative routes: HiDAR. In *Euroepan Symposium on Algorithms (ESA)*. Springer, 2013. to appear.

17 Moritz Kobitzsch, Dennis Schieferdecker, and Marcel Radermacher. Evolution and evaluation of the penalty method for alternative routes. In *ATMOS*, 2013.

18 Felix Koenig. Future challenges in real-life routing. In *Workshop on New Prospects in Car Navigation*. February 2012. TU Berlin.

19 Dennis Luxen and Dennis Schieferdecker. Candidate sets for alternative routes in road networks. In *Experimental Algorithms*, pages 260–270. Springer, 2012.

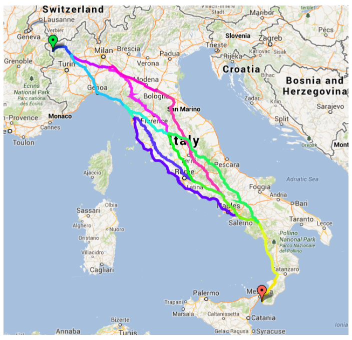
20 Georgia Mali, Panagiotis Michail, Andreas Paraskevopoulos, and Christos Zaroliagis. A new dynamic graph structure for large-scale transportation networks. In *Algorithms and Complexity*, volume 7878, pages 312–323. Springer, 2013.

21 Ernesto Queiros Vieira Martins. On a multicriteria shortest path problem. *European Journal of Operational Research*, 16(2):236–245, 1984.

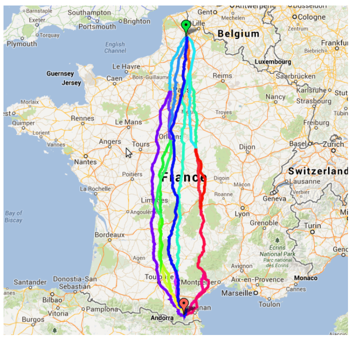
22 Jin Y Yen. Finding the k shortest loopless paths in a network. *management Science*, 17(11):712–716, 1971.

## A Appendix

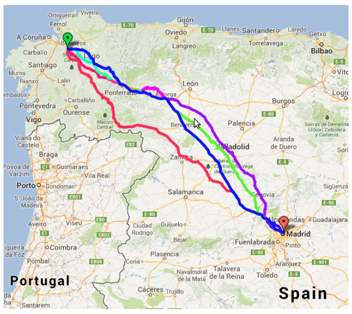
Figure 6 shows visualized AGs for a few representative cases.



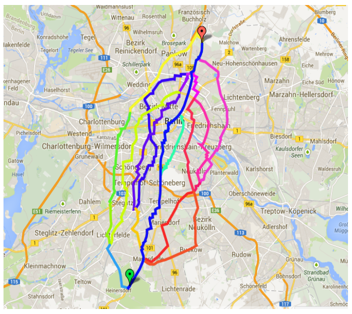
**(a)** Penalty



**(b)** Penalty and Plateau



**(c)** Plateau



**(d)** Penalty and Plateau

**Figure 6** Shape of AG: (a) Italy, (b) France, (c) Spain, (d) Berlin.

# Result Diversity for Multi-Modal Route Planning [*]

**Hannah Bast, Mirko Brodesser, and Sabine Storandt**

**Albert-Ludwigs-Universität Freiburg**
**Freiburg, Germany**
`{bast,brodessm,storandt}@informatik.uni-freiburg.de`

**Abstract**

We study multi-modal route planning allowing arbitrary (meaningful) combinations of public transportation, walking, and taking a car / taxi. In the straightforward model, the number of Pareto-optimal solutions explodes. It turns out that many of them are similar to each other or unreasonable. We introduce a new filtering procedure, *Types aNd Thresholds (TNT)*, which leads to a small yet representative subset of the reasonable paths. We consider metropolitan areas like New York, where a fast computation of the paths is difficult. To reduce the high computation times, optimality-preserving and heuristic approaches are introduced. We experimentally evaluate our approach with respect to result quality and query time. The experiments confirm that our result sets are indeed small (around 5 results per query) and representative (among the reasonable Pareto-optimal paths), and with average query times of about one second or less.

**1998 ACM Subject Classification** F.2.2 Nonnumerical Algorithms and Problems

**Keywords and phrases** Route Planning, Multi-Modal, Result Diversity

**Digital Object Identifier** 10.4230/OASIcs.ATMOS.2013.123

## 1 Introduction

We want to efficiently compute small yet representative sets of reasonable paths in a multi-modal scenario (car, walking, transit). The majority of current route planning systems computes optimal paths for a certain type of transportation. If one wants to take a car, one uses a navigation system. If one wants to travel by public transportation, one can obtain the optimal paths from websites like Google Maps. Both require to decide in advance for a means of transportation. For the case when one does not want to decide this beforehand, we want to offer the user a small yet representative set of reasonable paths. Moreover, we allow optimal paths which include different means of transportation. Furthermore, also for metropolitan areas like New York, computation should work fast, such that interactive queries are possible. For road networks, state-of-the-art algorithms answer shortest path queries in the order of milliseconds [9]. Public transportation networks are more challenging and many algorithms of road networks are not applicable [1]. Computing optimal paths becomes more complex, since not only the fastest connection is demanded, but also the number of transfers is an important criterion for the quality of a path, potentially leading to multiple optimal paths. When combining road and transit networks, this increases complexity further, bringing along many similar paths. Typical variations are: *"take a bus for 30 minutes, then take a taxi for 9 minutes"*, *"take a bus for 32 minutes, then take a taxi for 8 minutes"*, ..., *"take a bus for 42 minutes, then take a taxi for 3 minutes"*. To determine a small yet representative set of reasonable paths, it is necessary to filter, which is a challenge on its own. In the following we introduce an approach to deal with these challenges.

[*] Partially supported by a Google Focused Research Award.

### 1.1 Contribution

We propose *Types aNd Thresholds*, an approach to efficiently compute small yet representative sets of reasonable paths in a multi-modal scenario (car, walking, transit). To obtain diverse sets of paths, we use Pareto sets [11] with multiple optimization criteria. Taking into account properties (velocity, availability, costs) of the various means of transportation, we argue that not all Pareto optimal paths are reasonable. We carefully define types of reasonable paths and propose a two-stage filtering procedure. In the first stage, all unreasonable paths are removed. In the second stage, a small yet representative subset of the remaining paths is determined. To achieve average query durations of roughly one second, we make use of properties of the types and propose a relaxation of the model and a (close to optimal) heuristic. We confirm query durations and quality with experimental results.

## 2 Related Work

When considering road and transit networks separately, many algorithms exist for both networks. Next, we give a brief overview. For road networks several variants of the famous Dijkstra algorithm exist. An outstanding one is Contraction Hierarchies [9], which after a brief precomputation enables to quickly answer queries, even for large areas (e.g., Europe on the order of milliseconds). Fast routing on transit networks requires other approaches and algorithms. Among them are a multi-criteria generalization of Dijkstra's algorithm [8], relaxed Pareto dominance [13] and Round-Based Public Transit Routing [6]. One state-of-the-art algorithm is to compute transfer patterns [2] between all pairs of stations. A transfer pattern is the sequence of stations where vehicle changes occur on an optimal path. For each pair of stations these are few and allow to answer queries efficiently. However, the computation of patterns is time-consuming.

Also for multi-modal networks several approaches exist. However, many of those are quite limited with respect to the extent of their multi-modality. For instance, [5] limits car usage to the beginning and end of journeys. Other approaches [12] compute a single optimal path by combining multiple criteria to one, but this results in missing reasonable paths (see [3] for an example). Further approaches [16, 7] expect the user to specify a hierarchy of modes (e.g. between train usage, car is forbidden). The problem is that one has to know the constraints in advance, which in practice is often not the case.

A less restricted approach focussing on computing multiple optimal paths in a multi-modal scenario similar to ours was presented in [4]. To not miss reasonable paths, multiple criteria with Pareto sets [11] are used. As this leads to numerous optimal paths, fuzzy filtering is used to rank them according to scores. For the found set of paths $P = \{p_1, \dots, p_n\}$, the score of $p_i$ is dependent on $P$ and a measure for fractional dominance between a path $p \in P$ and the paths in $P \backslash \{p\}$. However, for the measures used in [4], the set of the top-$k$ paths is not necessarily representative when $k$ is small, and no experiments are provided on this quality aspect in [4]. In short, it is not clear if and how small yet representative sets of optimal paths can be determined with this approach.

## 3 Preliminaries

In this section, we describe how to model a multi-modal network and discuss the necessity for multiple optimality criteria. We briefly recapitulate Contraction Hierarchies as a speed-up technique and explain how to use existing algorithms to compute optimal paths in our setting.

## 3.1 Modelling

In the following we describe separate models for road and transit networks and how to combine them to a multi-modal model. To model the static road networks (car, walking), we use the common approach of one node per location (given as longitude, latitude) and time-independent arcs annotated with the duration to travel from one node to the other. For simplicity, we ignore turn restrictions and assume that all roads can be traveled in both directions by car and by foot. That is, the car and walking network have the same structure.

To model the transit network, we decided for a variation of the *train-route* model as explained in [15]. In the following we provide basic definitions and describe the model. A transit connection starting at a specific time at a specific station and ending at some station is called a *trip*. Trips sharing the same stop sequence and not overtaking each other are grouped as a *line*. For each stop, a *station arrival* and a *station departure* node are created. For each line, for each of its stops, a *line arrival* node and a *line departure* node are created. The nodes of a line are connected according to their stop sequence and the durations of the arcs are time-dependent. When boarding a line, the transfer buffer is added (we chose 5 minutes). Station arrival and station departure nodes are connected with their geographically closest car and walking node. We call these nodes *link* nodes. That is, each station arrival node has outgoing arcs to the closest car and walking node and they have outgoing arcs to the station departure node.

Unlike the model used in [4], our model prohibits to change from car to walking (and vice versa). The intention is that when going to a station by car, one does not stop on the way and walk the rest (or the other way around). Instead, to reach a station one either takes the car or walks. We consider this reasonable, since taking the car and walking is possible from and to all stations in our model. In practice, walking a short distance to or from a car is no problem, as we can consider this part of the transfer buffer. Note that car usage is *not* limited to the beginning and end of a journey, but is also allowed between taking two public transportation vehicles. We chose this model, because we consider it the most efficient in terms of query duration. Note that our filtering approach, which we describe in Section 4, is independent of the used model.

## 3.2 Optimality Criteria

In the following we explain the necessity for multiple optimality criteria to compute diverse sets of paths in our multi-modal scenario. When referring to *duration*, we mean the duration to reach the target, given a fixed departure time. Using duration as a single criterion would result in exactly one path. Therefore, we use multiple criteria with Pareto sets. Each criterion corresponds to one entry in a tuple. For tuples $t_1, t_2$, tuple $t_1$ is said to *dominate* $t_2$ if $t_1$ is at least as good as $t_2$ with respect to all criteria. Two tuples are called *incomparable* if none of them dominates the other. A Pareto set is a maximal set of non-dominating tuples. A *label* contains such a tuple and is associated with a predecessor label. In a multi-criteria Dijkstra, each node contains a Pareto set of labels.

For transit networks, duration and transfer penalty (= number of boarded vehicles) are two commonly used Pareto criteria. However, in our setting this almost always leads to exactly two optimal paths: walking the whole way and using the car for the whole way. The reasons are: Taking a car is very fast and in our model is available everywhere and boarding it once yields a transfer penalty of one, therefore it usually dominates all paths which include transit usage. Walking the whole way is incomparable to using the car, because it is slower and has zero transfers.

Therefore, we use car duration as another Pareto criterion, leading to a diverse set of optimal paths. However, solutions then become too numerous and many of them are similar. A typical variation is the one mentioned in the introduction, where paths differ only slightly in the car duration. In one of the scenarios from [4], they use arrival time (equivalent to our duration), number of transfers, walking duration, and taxi cost as Pareto criteria.

### 3.3 Contracting the Road Networks

As the majority of nodes are car and walking nodes (see Table 4 for details), optimizing the routing on the road network is important to reduce computation times. A well-known speed-up technique applicable to road networks is Contraction Hierarchies [9]. During shortest path queries it allows to skip many nodes, hence unnecessary propagation of labels is avoided. Recently, a variant [4] for a multi-modal scenario similar to ours was introduced, we refer to this as *contracting the road network*. Next, we summarize its most important properties. After an efficient precomputation, all nodes in the road network have a rank. There exists a *core* of nodes which comprises all link nodes and the rank of these nodes is infinity. For queries from all road network nodes the following holds: when ignoring arcs to nodes with lower rank, distances to all link nodes are equal to those in the original road network. As a special case distances between all pairs of link nodes are preserved. In the next section, we describe how to use these properties to efficiently compute shortest paths.

### 3.4 Computing Multi-Criteria Optimal Paths

In the following we explain how to perform location-to-location queries. We call the graph with inverted arc directions *backwards graph*. Given the road network is contracted as mentioned above (one contraction for usage by car, and one contraction for usage by walking), location-to-location queries are performed in two steps.

First, a query from the source to all nodes and a query (in the backwards graph) from the target to all nodes are performed. Recall that during the contraction process road network nodes were assigned a rank. For both queries, arcs to nodes with lower rank are ignored. We call the duration of walking (taking the car) from the source to the target, *pure* walking (car) duration.

Second, a multi-criteria Dijkstra initialized with the labels of the link nodes reached from the source is run. Temporary arcs with the previously computed durations from the link nodes to the target are added. Again, arcs to nodes with lower rank are ignored. The Pareto set of pure car and walking duration and the labels at the target forms the result. Dominance by early results and label forwarding [8] are used to accelerate query computation.

Note that this is essentially one of the query algorithms proposed in [4].

## 4 Types aNd Thresholds

In this section we describe the concept of *Types aNd Thresholds (TNT)* to obtain small but representative sets of reasonable paths. We present speed-up techniques to reduce query times towards practical usage. We start by introducing the idea of discretization which leads to TNT.

### 4.1 Discretization

Using duration, transfer penalty and car duration as Pareto criteria leads to numerous optimal paths, among which many are similar. To filter out a more concise subset, we

**Table 1** Excerpt of the tuples of the optimal paths of an example query using duration, transfer penalty and car duration as Pareto criteria. Green tuples are still Pareto optimal after the discretization, gray ones are not.

| duration | transfer penalty | car duration | discretized car duration |
|---|---|---|---|
| 0:28:57 | 1 | 0:28:57 | 0:30:00 |
| | | ... | |
| 1:43:43 | 3 | 0:16:35 | 0:20:00 |
| 1:44:01 | 3 | 0:16:26 | 0:20:00 |
| 1:44:09 | 3 | 0:16:04 | 0:20:00 |
| 1:44:34 | 5 | 0:11:07 | 0:20:00 |
| 1:44:36 | 3 | 0:15:56 | 0:20:00 |
| 1:45:12 | 4 | 0:15:51 | 0:20:00 |
| | | ... | |
| 7:06:00 | 0 | 0 | 0:00:00 |

examine the post-processing step of *discretizing* car duration to certain blocks (for example, ten minutes). Table 1 shows an excerpt of the results of a query on New York.

Discretization was also introduced in [4], however, it was used as a heuristic during query time to reduce computation complexity. Our motivation is different: given that many Pareto optimal solutions are similar, we use it to choose a representative subset.

Although discretization allows to reduce the number of Pareto-optimal solutions remarkably, unreasonable paths can remain. Consider the example in Figure 1. It is not very meaningful to walk a long distance and then take a taxi for a short distance. We argue that, in practice, one would either walk the whole way, walk and take a train or use the car for the whole way. In the following we propose a new approach to filter out such unreasonable paths, and then to obtain a small representative subset of the remaining paths.

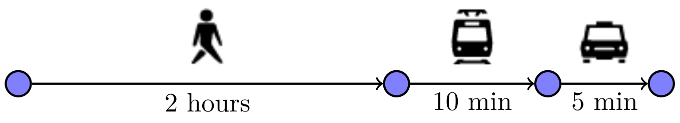


**Figure 1** An example for a path which we consider unreasonable.

## 4.2 Types

With the example in Figure 1, we illustrated that some Pareto optimal solutions can be unreasonable. To justify why certain types of paths are unreasonable, we analyze triples of transit, walking and car duration with respect to their *relative durations (RD)*. We classify each possible triple as either reasonable or unreasonable. As relative durations we use the abstract terms *zero* (○), *little* (⊙) and *much* (●), hence *RD* := {○, ⊙, ●}. In Section 4.3 we provide concrete definitions. We assume the natural order of ○ < ⊙ < ●. For example, the triple (●, ○, ⊙) represents paths with much transit usage, zero walking and little car usage.

Our rationale behind this (rather coarse) classification is as follows. Small differences in duration are of little practical concern to users, little is little. However, there is a difference between little and zero, because using a particular mode of transport at all incurs a tangible overhead (organizing a car/taxi, dealing with the circumstances of public transportation).

**Table 2** All combinations of relative durations (zero = ○, little = ⊙, much = ●) for transit, car and walking duration with the classifications and violated axioms. White background indicates a relative duration triple is not valid. From the remaining triples, the ones classified reasonable are green, the others are red.

| transit duration | walking duration | car duration | violated axiom | classification |
|---|---|---|---|---|
| ○ | ○ | ○ | | ✗ |
| ○ | ○ | ⊙ | | ✗ |
| ○ | ○ | ● | | ✓ |
| ○ | ⊙ | ○ | | ✗ |
| ○ | ⊙ | ⊙ | | ✗ |
| ○ | ⊙ | ● | A1 | ✗ |
| ○ | ● | ○ | | ✓ |
| ○ | ● | ⊙ | A2 | ✗ |
| ○ | ● | ● | A1 | ✗ |
| ⊙ | ○ | ○ | | ✗ |
| ⊙ | ○ | ⊙ | | ✗ |
| ⊙ | ○ | ● | A1 | ✗ |
| ⊙ | ⊙ | ○ | | ✗ |
| ⊙ | ⊙ | ⊙ | | ✗ |
| ⊙ | ⊙ | ● | A1 | ✗ |
| ⊙ | ● | ○ | | ✓ |
| ⊙ | ● | ⊙ | A2 | ✗ |
| ⊙ | ● | ● | A1 | ✗ |
| ● | ○ | ○ | | ✓ |
| ● | ○ | ⊙ | | ✓ |
| ● | ○ | ● | A1 | ✗ |
| ● | ⊙ | ○ | | ✓ |
| ● | ⊙ | ⊙ | | ✓ |
| ● | ⊙ | ● | A1 | ✗ |
| ● | ● | ○ | | ✓ |
| ● | ● | ⊙ | A2 | ✗ |
| ● | ● | ● | A1 | ✗ |

Once a certain mode of transportation is used more than little, it is reasonable to assume that one is willing to use it as much as is necessary to obtain an optimal solution. For example, if one is willing to use the car for one hour, one might as well use it for the whole trip if that is the fastest way. This is not necessarily true for walking (one might be willing to walk 1 hour but not 10 hours), however, that is not a problem in practice, because optimal paths rarely comprise very much walking (with the exception of the trivial walk-everything solution, which is always computed in our model).

As ⊙ and ● can be distinguished with respect to the total duration only if both occur in a triple, the set of all triples containing ⊙ but not ● is equivalent to the set of triples containing ● but not ⊙. Moreover, the triple without ⊙ and ●, that is (○, ○, ○), does not exist for real paths. Therefore, we call a triple *valid* iff at least one component is ●.

Consider the properties of our model and of the different modes of transportation:

- Public transit is limited to stations and schedules, medium-fast and medium-expensive.
- Walking is possible everywhere at all times, slow and cheap.
- Cars (taxis) are available everywhere at all times, fast and expensive.

Given these properties, we claim the following axioms should hold for all reasonable paths:

- A1: Much car usage implies zero walking and zero transit usage.
- A2: Much walking implies zero car usage.

From the axioms we deduce which triples of relative durations are reasonable. Table 2 contains all triples, annotated with the classification as *reasonable (✓)* or *unreasonable (✗).*

The classification is consistent in the sense that for each triple classified as reasonable, each valid component-wise smaller triple is classified reasonable, too. This can be inferred from Table 2. For instance, triple (●,○,⊙) is valid and triple (●,○,○), too. Finally, we determine three **types** incorporating all triples classified as reasonable:

1. Only car.
2. Much transit, much walking, no car.
3. Much transit, little walking, little car.

Here, the attributes *much* and *little* should be thought of to include the smaller relative durations. The types are complete to the effect that all relative duration triples classified as reasonable are included. This can again be deduced from Table 2. For practical purposes, relative durations need to be defined concretely. In the following, we introduce such definitions.

## 4.3 Thresholds

To practically use the types defined above, we propose to use **threshold** values for the relative durations. The following definitions reflect that *zero* signifies a transportation mode is not used, *little* depends on the mode and *much* means unlimited usage of a mode (durations in minutes):

- $zero(*) := 0$ min
- $little(walking) := 10$ min
- $little(car) := \begin{cases} 0 \text{ min}, & \text{if pure car duration} < 20 \text{ min} \\ \max(10 \text{ min}, 0.25 \cdot \text{pure car duration}), & \text{otherwise} \end{cases}$
- $much(*) := \infty$ min

Note that we defined the thresholds for a metropolitan setting. The definition for *much* is natural, since it represents everything which is greater than *little*. One can observe that the definition of *little(car)* is only relevant for paths belonging to type 3. Moreover, a path of this type is only interesting if it significantly differs from the path of type 1 (using the car for the whole way) in terms of car usage, otherwise one could just choose the path of type 1. Therefore, we chose 25% of its duration as upper bound but at least 10 minutes in order to avoid enforcing absurdly low car durations. For *little(walking)* we decided for a fixed threshold in order to allow nearby stations to be reached but avoiding journeys where walking significantly exceeds car usage. We consider a fixed threshold reasonable, as paths of type 3 have a duration of at most a few hours (in a metropolitan setting). If we chose *little(walking)* dependent on this maximal duration, it would be bounded from above by an absolute value anyway. However, we want to stress that the types can be used with other definitions of thresholds as well.

## 4.4 Filtering

We introduce a post-processing procedure to obtain small yet representative sets of reasonable paths from Pareto sets. Given the above defined types and (arbitrary) thresholds, we explain how to use them to remove unreasonable paths and how to obtain small yet representative paths in a second step. We call the whole concept **Types aNd Thresholds (TNT)**.

We say a path *belongs to a type* if none of the type's thresholds is exceeded. For instance, assuming a pure car duration of 15 minutes, the path of Figure 1 belongs to none of the three types. To remove all unreasonable paths, the ones belonging to no type are removed. Note that for optimal results, walking duration must be considered as a Pareto criterion, too. That is, Pareto criteria are duration, transfer penalty, car duration and walking duration.

**Table 3** Excerpt of the tuples of the optimal paths for an example query for Dallas. Pareto criteria are duration, transfer penalty, car duration and walking duration. Green tuples remained after filtering with TNT, gray ones did not. Before filtering, there were 66 Pareto optimal paths, after filtering only 7 reasonable paths remain.

| duration | transfer penalty | walking duration | car duration | type |
|---|---|---|---|---|
| 0:29:17 | 1 | 0:00:00 | 0:29:17 | 1 |
| | | ... | | |
| 1:52:11 | 4 | 0:07:33 | 0:13:35 | none |
| 1:56:10 | 4 | 0:04:18 | 0:09:52 | 3 |
| 1:56:10 | 5 | 0:06:54 | 0:09:35 | 3 |
| | | ... | | |
| 2:08:49 | 3 | 0:02:17 | 0:09:43 | 3 |
| 2:42:13 | 3 | 0:48:42 | 0:00:00 | 2 |
| 2:57:49 | 2 | 0:54:38 | 0:00:00 | 2 |
| 3:37:10 | 1 | 2:23:07 | 0:00:00 | 2 |
| 6:02:31 | 0 | 6:02:31 | 0:00:00 | 2 |

After removing the unreasonable paths, we drop walking duration as a Pareto criterion. This removes undesired (minor) variation in the result set with respect to walking duration. Potentially, this can lead to the loss of interesting (reasonable) optimal paths. However, that is unlikely because it is unlikely that a path with higher walking duration dominates a path with lower walking duration in all other criteria.

To determine a small and representative subset from the remaining paths, we propose to transform all car durations according to their relative durations:

$$rd(\text{car duration}) := \begin{cases} 0, & \text{if car duration} = \text{zero(car)} \\ 1, & \text{if zero(car)} < \text{car duration} \leq \text{little(car)} \\ 2, & \text{if little(car)} < \text{car duration} < \text{much(car)} \end{cases}$$

Note that this coarse discretization is in sync with our coarse classification of travel times into three categories (zero, little, much) argued for in Section 4.2. The Pareto set of the transformed labels constitutes the result, now indeed a small yet representative subset of the reasonable Pareto-optimal solutions. Table 3 shows an excerpt of the results of a real query for Dallas.

**Influence of Thresholds on the Results.** The choice of fixed thresholds obviously restricts the space of possible paths, but one does not want to miss significantly better paths which do not severely exceed the thresholds. Next, we explain how this can be achieved.

An advantage of our system is that lowering the thresholds can never lead to better paths with respect to duration and transfer penalty. To avoid missing significantly better paths which are not severely above the thresholds we propose to offer the user an outlook of how the paths improve with a higher threshold. One way to achieve this is to compute the difference in duration of the fastest paths for two significantly different thresholds. This enables the user to decide if she wants to run another query with modified threshold values.

## 4.5 Speed-up Techniques for Faster Query Answering

Each additional Pareto criterion enlarges the set of optimal paths significantly, resulting in infeasible query times for large datasets. To speed up query computation we introduce an optimality preserving extension, a relaxation of the model and a heuristic.

### 4.5.1 Extended Dominance by Early Results

To prune labels during query computation dominance by early results was introduced in [8]. All labels dominated by the target labels can be discarded, since they and all their extensions can not become Pareto-optimal at the target. For TNT, this can be extended to labels not belonging to any type. Extensions of such labels can not belong to any type and can therefore be discarded. Therefore, for the multi-criteria Dijkstra, dominance by early results can be extended to ignore labels for which the following holds:

- *walking duration* $>$ *little(walking)* and *car duration* $>$ *zero(car)* or
- *car duration* $>$ *little(car)*

Recall that *little(car)* depends on pure car duration, which is computed in the first step of the shortest-path computation described in Section 3.4. Hence, it is available for the (time-consuming) multi-criteria Dijkstra. Note that this optimization is applicable independent from the threshold definition.

### 4.5.2 Rounding on Transfers

As we exemplified in Table 1, many Pareto optimal paths are similar. The table indicates that many solutions differ only by seconds. In our implementation, arc durations are stored with a resolution of one second. This is enough to avoid the accumulation of rounding errors (which for a resolution of, say, one minute, would tangibly impact result optimality). For road networks, we calculate durations depending on distances and speed, leading to an accuracy of seconds. The GTFS data [10], which we use to model the transit network, provides durations in seconds.

However, public transportation in practice rarely provides accuracy by seconds and the speed of humans in terms of walking and car usage varies, too. Inspired by discretization, we propose to relax the model and *to round up durations to full minutes immediately before transfers*. Compared to rounding at each node, this restricts error accumulation sufficiently. Moreover, it can be interpreted as a coarse transfer buffer. With respect to reality, we consider this an optimality preserving technique.

Arc relaxations will happen for less labels, and less comparisons have to be performed when inserting a label to the set of labels attached to a node. Therefore, we expect rounding during query time to notably speed-up computation time.

### 4.5.3 Using Implicit Walking Duration

As mentioned in Section 4.4, when filtering labels to their types, walking duration has to be a Pareto criterion to obtain optimal results. Nevertheless, when walking duration is not a Pareto criterion, we expect the difference to the optimal results to be minor. Therefore, we propose the heuristic of using implicit walking duration by keeping it in a hidden variable, which is not used as Pareto criterion. The priority queue order is chosen such that in case of tie-breaking the label with less walking duration is released earlier from the queue. For labels with equal Pareto criteria the one with less walking duration is kept. It is worth noting that using implicit walking duration does not affect the quality of type 1 (only car) and type 2 (much transit, much walking, no car) paths. For type 1, this is clear since the optimal path is computed separately using only duration as criterion. As labels of type 2 (that is, with car duration $= 0$) cannot be dominated from labels with car duration $> 0$, we can ignore car duration when proving the optimality for labels of type 2 in the following lemma.

▶ **Lemma 1.** *Let $L_A$ be the label set at a node $u \in V$ after termination of the Pareto-Dijkstra run considering the criteria D(uration), T(ransfer) P(enalty) and W(alking) D(uration). Also let $L_B$ be the respective label set for a Pareto-Dijkstra run regarding only the criteria D and TP. We claim that $\forall l \in L_A \exists l' \in L_B : l' \leq l \mid_{(D,TP)}$.*

**Proof.** Let $l^* \in L_A$ be the label with $TP(l^*) = TP(l)$ and minimal duration. Obviously when neglecting walking duration and following the same path that lead to the creation of $l^*$ at $u$, the label $l^* \mid_{(D,TP)}$ is a possible candidate for being in $L_B$. Hence it must exist a label $l' \in L_B$ with $TP(l') \leq TP(l^*)$ and $D(l') \leq D(l^*)$. Therefore it holds $l' \leq l^* \mid_{(D,TP)} \leq l \mid_{(D,TP)}$. ◀

To see that paths of type 3 are not necessarily optimal, consider the following tuples which are incomparable with Pareto criteria duration, transfer penalty, car and walking duration: (40 min, 2, 10 min, 5 min) and (30 min, 2, 10 min, 6 min). Using implicit walking duration, only the latter would be optimal. However, assuming an extension by 5 minutes of walking, the latter tuple would belong to no type and hence be filtered out, whereas the former would not.

## 5 Experimental Results

In this section we evaluate the concept of using Types aNd Thresholds (TNT) and its speed-up techniques with respect to result quality and query time.

### 5.1 Setup

Our implementation of the graph model and the optimal path algorithm, as described in Section 3, is written in `C++` and compiled with `GCC 4.6.3` with the `-O3` flag. Experiments were performed on a machine with 96GB of RAM and two Intel Xenon E5649 CPUs with 8 cores, each having a frequency of 2.53 GHz (exactly one core was used at a time). The used OS is Ubuntu `12.04`, operating in 64-bit mode.

To instantiate the multi-modal networks we used publicly available OSM [14] and GTFS data [10]. Details on our data sets can be found under `http://ad.informatik.uni-freiburg.de/publications`. We used the data of the first available Monday. OSM data was chosen to cover the terrain corresponding to the GTFS data. The road network graphs are symmetric and reduced to their largest connected component. For walking, we assumed an average speed of 5 km/h. For the car network, average velocity was chosen depending on the road type, ranging from 5 to 110 km/h. We evaluated our algorithm on the networks of *Austin, Dallas, Toronto* and *New York City* (in the following abbreviated as just *New York*). Table 4 contains an overview of the most important properties of these networks.
For each dataset, experiments were performed using 1000 queries between two random locations, each at most 1 km away from at least one transit station. We chose this restriction to avoid a significant amount of queries in areas where transit is not available, in which case the only interesting solutions would be car-only and walking-only. Departure times were chosen uniformly at random from the time range between 6:00 a.m. and 10:00 p.m.

### 5.2 Results

We evaluate the concept of TNT with respect to query time and quality of the found sets of paths. Experiments were performed for the normal graph model (Section 3.1) and the model with rounding on transfers (Section 4.5.2). We refer to the latter as the *relaxed* model. For

**Table 4** Overview of important properties of the evaluated networks. Recall that the time-consuming step of the path computation operates on the cores instead of the whole road networks.

| Graph | | Austin | Dallas | Toronto | New York |
|---|---|---|---|---|---|
| Complete | Nodes | 0.7M | 2.8M | 0.9M | 4.0M |
| | Arcs | 2.9M | 12.0M | 3.7M | 17.3M |
| Transit | Stations | 2.7K | 11.6K | 10.9K | 16.9K |
| | Nodes | 14.3K | 49.0K | 80.4K | 118.6K |
| | Arcs | 21.2K | 73.0K | 119.5K | 175.9K |
| | Lines | 235 | 563 | 1120 | 1989 |
| | Trips | 6062 | 10849 | 40740 | 62824 |
| Car | Nodes (core) | 3.3K | 20.8K | 10.9K | 28.1K |
| Walking | Nodes (core) | 4.0K | 20.2K | 12.2K | 32.5K |

**Table 5** Average query times for all datasets.

| Model | Algo | Austin | Dallas | Toronto | New York |
|---|---|---|---|---|---|
| Normal | Basic | 0.6s | 3.7s | 16.6s | 108.0s |
| | IWD | 0.1s | 0.8s | 0.6s | 1.7s |
| Relaxed | Basic | 0.4s | 2.2s | 4.0s | 18.0s |
| | IWD | 0.1s | 0.8s | 0.6s | 1.4s |

both models we compare the basic algorithm (Section 3.4) and the heuristic of using implicit walking duration (IWD, Section 4.5.3).

Table 5 shows average query times for all of our four datasets.

It indicates that both, rounding on transfers and the IWD heuristic reduce query times. While the heuristic has a stronger effect, the lowest query times (roughly one second) are achieved by applying both. It is noteworthy that the speed-up increases significantly with the size of the network (roughly from factor 5 to factor 75). Query times are comparable to those presented in [4].

For the largest dataset (New York) we evaluate the basic algorithm and the IWD heuristic in more detail, for both models and with respect to both result quality and query time; see Table 6. As quality measures of the heuristic we use precision[1] and recall[2]. It can be seen that (with the IWD heuristic in the relaxed model) for a few outliers the query time rises up to seven seconds, but the majority of queries can be answered in roughly one second. Precision and recall indicate that for both models the IWD heuristic leads to only a small fraction of non-optimal results.

To evaluate if the computed sets of paths are small and representative, Table 7 shows the distribution of paths with respect to our types for the basic algorithm on New York. Paths of type 3 that also belong to type 2 were only counted for type 3. For example, the table shows that around 15.8% of the queries lead to exactly one path of type 1, three paths of type 2 and one path of type 3. Note that for almost 50% of the queries there is no optimal path of type 3. One reason for this is that if pure car duration is relatively small (below 20 minutes, see section 4.3 and 4.4), no path solely belonging to type 3 can exist.

To see how paths of type 3 improve when increasing the *little(car)* threshold, we experi-

[1] Precision = $|relevant\text{-}paths \cap found\text{-}paths| \ / \ |found\text{-}paths|$
[2] Recall = $|relevant\text{-}paths \cap found\text{-}paths| \ / \ |relevant\text{-}paths|$

**Table 6** Query times and result quality for New York. For all measured variables we list average, 50-percentile, 90-percentile and 99-percentile values.

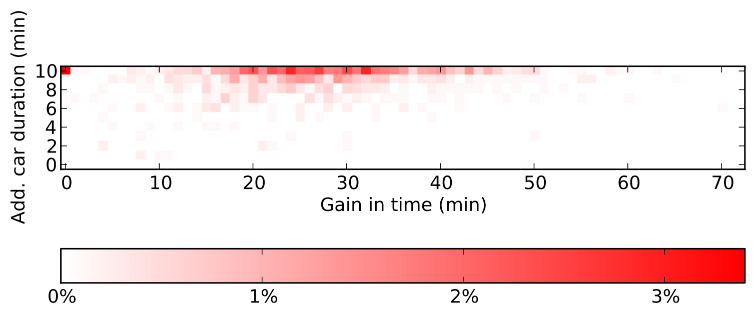
| | | Time [s] | | | | Precision | | | | Recall | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Model | Algo | avg | 50 | 90 | 99 | avg | 50 | 90 | 99 | avg | 50 | 90 | 99 |
| Normal | Basic | 108.0 | 32.7 | 289.0 | 865.0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| | IWD | 1.7 | 1.0 | 3.3 | 10.0 | 0.99 | 1 | 1 | 1 | 0.96 | 1 | 1 | 1 |
| Relaxed | Basic | 18.0 | 8.8 | 40.0 | 140.0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| | IWD | 1.4 | 1.0 | 2.5 | 6.6 | 0.99 | 1 | 1 | 1 | 0.96 | 1 | 1 | 1 |

**Table 7** Percentage of queries which lead to the different combinations of paths of type 2 and type 3. For each query one path of type 1 was optimal.

| #type-3 paths | | | | | | | |
|---|---|---|---|---|---|---|---|
| 3 | - | - | 0.1% | 1.0% | 0.2% | - | - |
| 2 | - | 0.9% | 5.6% | 7.0% | 2.2% | 0.4% | - |
| 1 | - | 1.2% | **15.8%** | **14.4%** | 2.7% | 0.3% | 0.1% |
| 0 | 1.6% | **10.6%** | **20.9%** | **12.5%** | 2.4% | 0.1% | - |
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| | #type-2 paths | | | | | | |

mentally evaluated the maximal gain in time when extending the threshold by 10 minutes. For this, we considered queries which already for *little(car)* had labels of type 3 and compared the fastest such label (= with the smallest duration) with the fastest label when using the extended threshold. Figure 2 shows the results for New York. For 42% of the queries, increasing car travel time up to 10 minutes allows to reduce the total duration by 20-30 minutes. For practically every query increasing *little(car)* leads to faster paths. As explained in Section 4.4, this information could be communicated to the user (for the given query), with the option to relaunch the query with an accordingly modified threshold value.

**Figure 2** Maximal possible gain in time for labels of type 3, when allowing additional car travel time of up to 10 minutes. The heat map shows the gain in time and respective additional car travel time (with respect to *little(car))* for the different queries.



## 6 Conclusions & Future Work

We studied multi-modal route planning involving (almost) unrestricted combinations of walking, car, and transit. The goal was to efficiently compute small yet representative sets of optimal paths. We illustrated that multiple criteria are necessary to obtain diverse sets of paths. To remove unreasonable paths and to extract a small representative subset of the remaining paths, we introduced a new approach of using Types aNd Thresholds (TNT). To reduce infeasible query times induced by multiple optimality criteria, we introduced an extension of dominance by early results, a relaxation of the model (rounding on transfers) and a heuristic. We experimentally evaluated TNT and the speed-up techniques. While the basic algorithm results in infeasible query times, relaxing the model and using the (almost optimal) heuristic reduces them to an average of roughly one second. Our experiments confirmed that our result sets are indeed small and representative, at least from the point of view of our model. Possible future work comprises examining other threshold definitions, extending the filtering step and considering fare zones. Lowering query times when using TNT is a further challenge. For the latter, one possibility could be to extend Transfer Pattern Routing [2] to our multi-modal scenario. Moreover, reliability and robustness (i.e., if connections are missed, how good are the alternatives) are important issues to consider.

**Acknowledgements** We want to thank an anonymous reviewer for his/her extensive feedback.

## References

1 Hannah Bast. Car or public transport – two worlds. In *Efficient Algorithms, LNCS 5760*, pages 355–367, 2009.

2 Hannah Bast, Erik Carlsson, Arno Eigenwillig, Robert Geisberger, Chris Harrelson, Veselin Raychev, and Fabien Viger. Fast routing in very large public transportation networks using transfer patterns. In *ESA, LNCS 6346*, pages 290–301, 2010.

3 Daniel Delling, Julian Dibbelt, Thomas Pajor, Dorothea Wagner, and Renato F. Werneck. Computing and evaluating multimodal journeys. Technical report, Karlsruhe Institute of Technology, 2012.

4 Daniel Delling, Julian Dibbelt, Thomas Pajor, Dorothea Wagner, and Renato F. Werneck. Computing multimodal journeys in practice. In *SEA, LNCS 7933*, pages 260–271, 2013.

5 Daniel Delling, Thomas Pajor, and Dorothea Wagner. Accelerating multi-modal route planning by access-nodes. In *ESA, LNCS 5757*, pages 587–598, 2009.

6 Daniel Delling, Thomas Pajor, and Renato Fonseca F. Werneck. Round-based public transit routing. In *ALENEX*, pages 130–140, 2012.

7 Julian Dibbelt, Thomas Pajor, and Dorothea Wagner. User-constrained multi-modal route planning. In *ALENEX*, pages 118–129, 2012.

8 Yann Disser, Matthias Müller-Hannemann, and Mathias Schnee. Multi-criteria shortest paths in time-dependent train networks. In *WEA, LNCS 5038*, pages 347–361, 2008.

9 Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *WEA, LNCS 5038*, pages 319–333, 2008.

10 General Transit Feed Specification (GTFS). `https://developers.google.com/transit/gtfs/`, October 2012.

11 P. Hansen. Bricriteria path problems. In *Fandel, G., Gal, T. (eds.) Multiple Criteria Decision Making – Theory and Application*, pages 109–127. Springer, 1979.

12 Paola Modesti and Anna Sciomachen. A utility measure for finding multiobjective shortest paths in urban multimodal transportation networks. *European Journal of Operational Research*, 111(3):495–508, 1998.

13 Matthias Müller-Hannemann and Mathias Schnee. Finding all attractive train connections by multi-criteria pareto search. In *ATMOS, LNCS 4359*, pages 246–263, 2004.

14 Open Street Map (OSM). `http://www.openstreetmap.org`, October 2012.

**15** Evangelia Pyrga, Frank Schulz, Dorothea Wagner, and Christos D. Zaroliagis. Efficient models for timetable information in public transportation systems. *ACM Journal of Experimental Algorithmics*, 12, 2007.

**16** Haicong Yu and Feng Lu. Advanced multi-modal routing approach for pedestrians. In *Consumer Electronics, Communications and Networks (CECNet), 2012 2nd International Conference on*, pages 2349–2352, April 2012.

# Column Generation for Bi-Objective Vehicle Routing Problems with a Min-Max Objective

Boadu Mensah Sarpong[1,2], Christian Artigues[2,3], and Nicolas Jozefowiez[1,2]

1 CNRS, LAAS, 7 avenue du colonel Roche, F–31400 Toulouse, France
{bmsarpon,artigues,njozefow}@laas.fr

2 Université de Toulouse, INSA, LAAS, F–31400 Toulouse, France

3 Université de Toulouse, LAAS, F–31400 Toulouse, France

## Abstract

Column generation has been very useful in solving single objective vehicle routing problems (VRPs). Its role in a branch-and-price algorithm is to compute a lower bound which is then used in a branch-and-bound framework to guide the search for integer solutions. In spite of the success of the method, only a few papers treat its application to multi-objective problems and this paper seeks to contribute in this respect. We study how good lower bounds for bi-objective VRPs in which one objective is a min-max function can be computed by column generation. A way to model these problems as well as a strategy to effectively search for columns are presented. We apply the ideas to two VRPs and our results show that strong lower bounds for this class of problems can be obtained in "reasonable" times if columns are intelligently managed. Moreover, the quality of the bounds obtained from the proposed model are significantly better than those obtained from the corresponding "standard" approach.

**1998 ACM Subject Classification** G.1.6 Integer Programming, G.2.3 Applications

**Keywords and phrases** multi-objective optimization, column generation, integer programming, vehicle routing

**Digital Object Identifier** 10.4230/OASIcs.ATMOS.2013.137

## 1 Introduction

Bounds (lower and upper) have been the backbone of methods for solving difficult single objective problems including VRPs. For this reason, it is natural to expect that bounds will also be useful for multi-objective problems. It is, thus, necessary to develop models and strategies for computing good bounds for multi-objective problems. In this paper, we study the use of column generation in computing strong lower and upper bounds for bi-objective VRPs in which one objective is a min-max function. We will use the acronym BOVRPMMO to refer to a problem of this kind.

A BOVRPMMO can be defined by means of a Dantzig-Wolfe decomposition as the selection of a set of columns with minimum total cost such that the maximum value of an attribute associated with the set is minimized. More formally, we consider problems of the form:

$$\begin{aligned}
&\text{Minimize} \quad \sum_{k\in\Omega} c_k\lambda_k && (1)\\
&\text{Minimize} \quad \Gamma_{\max} && (2)\\
\text{subject to} \quad &\sum_{k\in\Omega} a_{ik}\lambda_k \;\geq\; b_i \qquad (i\in I)\;, && (3)\\
&\Gamma_{\max} \;\geq\; \sigma_k\lambda_k \qquad (k\in\Omega)\;, && (4)\\
&\lambda_k \;\in\; \{0,1\} \qquad (k\in\Omega)\;, && (5)
\end{aligned}$$

where $\lambda_k$ and $\Gamma_{\max}$ are decision variables, $\Omega$ is the set of all feasible columns whose description depends on the particular problem, and $I$ is an index set. For each column $k \in \Omega$, $c_k$ and $\sigma_k$ are two associated values which we suppose to be integers. We need to select columns with minimum sum of $c_k$ such that $\Gamma_{\max} = \max_{k\in\Omega}\{\sigma_k\lambda_k\}$ is also minimized. Bi-objective generalizations of several vehicle routing problems satisfying this condition can be defined. In general, we want to minimize the combined cost of a set of routes such that the value of a property associated with the selected routes (eg. the maximum length of a route, max capacity of a route, etc.) is minimized. We will later present two of such problems namely the bi-objective uncapacitated vehicle routing problem (BOUVRP) and the bi-objective multi-vehicle covering tour problem (BOMCTP).

A BOVRPMMO is a special case of a multi-objective combinatorial optimization (MOCO) problem. A general MOCO problem concerns the minimization of a vector of two or more functions $F(x) = (f_1(x), \ldots, f_r(x))$ over a finite domain of feasible solutions $\mathcal{X}$. The vector $x = (x_1, \ldots, x_n)$ is the decision variable or solution, $\mathcal{Y} = F(\mathcal{X})$ corresponds to the images of the feasible solutions in the objective space, and $y = (y_1, \ldots, y_r)$, where $y_i = f_i(x)$, is a point of the objective space. A solution $x'$ *dominates* another solution $x''$ if for any index $i \in \{1, \ldots, n\}$, $f_i(x') \leq f_i(x'')$ and there is at least one index $i \in \{1, \ldots, n\}$, such that $f_i(x') < f_i(x'')$. A feasible solution dominated by no other feasible solution is said to be *efficient* or *Pareto optimal* and its image in the objective space is said to be *nondominated*. The set of all efficient solutions is called the *efficient set* (denoted $\mathcal{X}_E$) and the set of all nondominated points is the *nondominated set* (denoted $\mathcal{Y}_N$). Although the meaning of bounds in single objective optimization is well studied and understood, the situation is quite different in the multi-objective case. Ideal and nadir points are well known lower and upper bounds, respectively, of the set $\mathcal{Y}_N$. The coordinates of the ideal point are obtained by optimizing each objective function independently of the others, whereas the coordinates of the nadir point correspond to the worse value of each objective function when we consider the set $\mathcal{X}_E$. From Figure 1, it can be seen that these points are usually poor bounds since they just estimate the whole region where a member of $\mathcal{Y}_N$ may lie. In this paper, we will be interested in bounds that can reduce the region where the members of $\mathcal{Y}_N$ are and thus narrow down the search for nondominated points.

Given that a MOCO problem is a discrete problem, its lower bound can be defined as a finite set of points such that the image of every feasible solution is dominated by at least one of these points [15]. The members of a lower bound set do not necessarily belong to $\mathcal{Y}$. An upper bound may also be defined as a finite set of points in $\mathcal{Y}$ that do not dominate one another. This idea of *bound sets* for bi-objective combinatorial optimization (BOCO) problems has recently been revisited by other authors [3, 6, 14]. They compute strong lower bounds based on a weighted sum scalarization which can then be used in developing exact algorithms for the problems they consider. Although each objective function $f_i$ of the vector $F$ can either be a *sum objective* as in (1) or a *min-max objective*, examples given in the

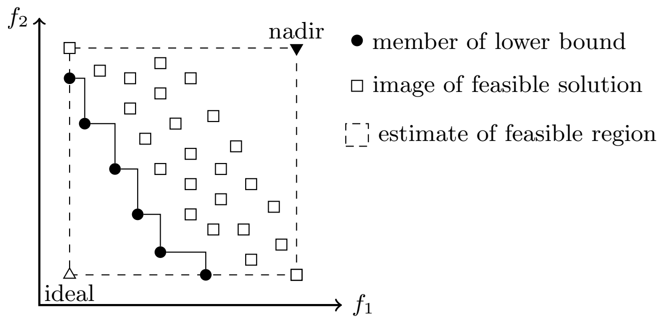


**Figure 1** Lower and upper bounds of a bi-objective combinatorial optimization problem.

cited papers consider only the "sum" type of objectives. This in a way justifies the use of a weighted sum method since they can efficiently find supported efficient solutions (those that correspond to points on the convex part of $\mathcal{Y}_N$) by using well known single objective optimization methods. Very good lower bounds for many problems can be defined from the set of supported efficient solutions. The situation is quite different when we consider a combination of a sum and a min-max objective function as in the case of a BOVRPMMO. Indeed, a general linearizing method for the min-max objective destroys the problem structure and so the desirable quality of being able to use known methods for the resulting problem does not necessarily apply [5]. A similar thing happens when we use a standard $\varepsilon$-constraint approach although this latter approach can find non-supported solutions which cannot be found by the weighted sum method. Moreover, as shown by results in [6], the quality of the bounds obtained by the weighted sum method for set covering problems are not very good. Nevertheless, the quality of lower bounds produced from set covering based formulations for single objective VRPs are one the best and so we can expect good quality lower bounds from formulations of this type in the multi-objective case too. these reasons, the approach proposed in this paper uses a variant of the $\varepsilon$-constraint method applied to a set covering based formulation for the BOVRPMMO.

The main contribution of this work is to define the application of column generation to a BOVRPMMO that does not rely on any "standard" multi-objective technique. In this way, we avoid some drawbacks such as the impossibility to find non-supported solutions by a weighted sum method or the possible loosening of a lower bound by explicitly adding constraints on objectives as it in the case of a standard $\varepsilon$-constraint approach. Moreover, if the problem linked to the first objective is a well-studied problem for which an efficient column generation algorithm exists then it is possible to reuse the pricing scheme to solve the bi-objective problems obtained by adding a second objective linked to a property of a selected set of columns. Another contribution is the computation of bound sets by using a scalar method other than the weighted sum method which has been used in published papers (to the best of our knowledge).

The use of column generation to compute bounds for a BOVRPMMO is explained in Section 2. Applications problems are discussed in Section 3. Computational results and conclusions are provided in Sections 4 and 5, respectively.

## 2 Column Generation for a BOVRPMMO

A close examination of formulation (1 – 5) reveals that a BOVRPMMO decomposes naturally into two problems. For any set of feasible columns, the associated value of $\Gamma_{\max}$ can easily be computed. We can therefore use a variant of the $\varepsilon$-constraint method with one main difference. Instead of explicitly adding a constraint of the form $\Gamma_{\max} \leq \varepsilon$ to the formulation, we rather drop (4) and use it to redefine the feasibility of a column. Thus, we define a new set of feasible columns $\bar{\Omega}$ where the feasibility of a column $k \in \bar{\Omega}$ now depends on its associated value $\sigma_k$. Depending whether or not a column $k \in \Omega$ may be associated with more than one value of $\sigma_k$, we may have a larger set of feasible columns after the redefinition. The strength of the model is conserved at the expense of having a possibly more difficult problem due to the possible increase in the number feasible columns. The master problem (MP) becomes the following single-objective program:

$$\text{Minimize} \sum_{k\in\bar{\Omega}} c_k\lambda_k \tag{6}$$

$$\text{subject to} \quad \sum_{k\in\bar{\Omega}} a_{ik}\lambda_k \geq b_i \qquad (i \in I)\ , \tag{7}$$

$$\lambda_k \in \{0,1\} \quad (k \in \bar{\Omega})\ . \tag{8}$$

Before solving MP for a given limit $\varepsilon$ on the value of $\Gamma_{\max}$, we need to set $\lambda_k = 0$ for all columns $k \in \bar{\Omega}_k$ having $\sigma_k > \varepsilon$. The linear relaxation of MP (ie. $\lambda_k \geq 0\ \forall k \in \bar{\Omega}$) is denoted as LMP.

## 2.1 Computing Lower and Upper Bounds

For a BOVRPMMO, $\Gamma_{\max}$ can only take on a finite number of values. If the complete set of feasible columns $\bar{\Omega}$ is known, a lower bound can be computed by using a variant of the $\varepsilon$-constraint approach as given in Algorithm 1. The algorithm starts with no restriction on the value of $\sigma_k$ for a column. At each iteration, a linear relaxation of the problem is solved after which the optimal value as well as the value of $\Gamma_{\max}$ are determined. In the next iteration, the problem is updated to exclude columns $k$ for which $\sigma_k$ is greater than $\Gamma_{\max}$. This iterative process continues for as long as the problem remains feasible. In practice, the cardinality of $\bar{\Omega}$ is too large and so a column generation method needs to be used by considering only a subset $\bar{\Omega}_1$ of $\bar{\Omega}$. The restriction of MP to $\bar{\Omega}_1$ is called the restricted master problem (RMP) and the resulting linear relaxation (ie. $\lambda_k \geq 0\ \forall k \in \bar{\Omega}_1$) is denoted LRMP. Let $\pi_i$ $(i \in I)$ be the dual variables associated with LRMP. The subproblem is defined as

$$S(\varepsilon) = \min_{k\in\bar{\Omega}\setminus\bar{\Omega}_1} \left\{ c_k - \sum_{i\in I} \pi_i a_{ik} : \sigma_k \leq \varepsilon \right\}\ , \tag{9}$$

where $\varepsilon$ is a maximum allowed value of $\Gamma_{\max}$ in LRMP. In applying column generation to compute a lower bound, we need to be able to efficiently search for relevant columns and so we explore some strategies in the next subsection.

**Algorithm 1** Computing a lower bound

```
1: Set $lb \leftarrow \emptyset$.
2: while LMP is feasible do
3:     Solve LMP. Let $c^*$ be the optimum, and $\lambda^*$ be the optimal solution vector.
4:     Compute $\Gamma_{\max} = \max_{k\in\bar{\Omega}} \sigma_k\lambda^*_k$.
5:     $lb \leftarrow lb \cup \{(c^*, \Gamma_{\max})\}$.
6:     Set $\lambda_k \leftarrow 0$ for all $k$ such that $\sigma_k \geq \Gamma_{\max}$.
7: end while
```

After computing a lower bound, a simple way to compute an upper bound is to solve RMP (the integer program) several times by following the idea of Algorithm 1. That is, we consider the RMP with the columns it contains after computing a lower bound and follow Algorithm 1 by replacing LMP with RMP. This is perhaps the simplest column generation heuristic. Although an upper bound is made up of feasible points, they do not necessary belong to $\mathcal{Y}_N$ since the RMP may not contain all relevant columns. Nevertheless, if the columns in RMP are relevant for the integer program then we expect that the upper bound produced will be a good approximation of $\mathcal{Y}_N$.

## 2.2 Column Search Strategies

A first approach of applying column generation to compute a lower bound of a BOIPMMO follows the idea of a standard $\varepsilon$-constraint method. For a fixed value of $\varepsilon$, LRMP is solved to optimality by column generation before moving on to another value of $\varepsilon$. An iteration of column generation consists in solving LRMP once to obtain a vector of dual values, and then solving the corresponding subproblem to obtain new columns to add to LRMP. The method converges when no new columns are produced from the subproblem. We denote this approach as the "Point-by-Point Search (PPS)". Although PPS is simple and easy to implement, it takes no advantage of the similarities in the subproblems for the different values of $\varepsilon$ and so a tremendous amount of computational time may be spent in computing each member of a lower bound.

Using heuristics to generate columns can improve the performance of column generation [4]. These heuristics are used to cheaply generate other relevant columns from those found by a subproblem algorithm. In the bi-objective case, we are interested in heuristics that can take advantage of similarities in the different subproblems solved when computing each point of a lower bound. That is, once the cost of finding a first column has been paid we wish to quickly generate other relevant columns that are relevant for the current subproblem and may also be relevant for other subproblems. A column which has negative reduced cost for a current subproblem, does not necessary have a negative reduced cost for another subproblem since the associated dual variables do not necessarily have the same values. Nevertheless, it can be expected that two subproblems that are close in terms of objectives, may also be close in terms of the solution of LRMP and therefore close in terms of dual variable values. For this reason, a column generated by a heuristic may also be of negative reduced cost for several other subproblems apart from the current one. In addition, standard algorithms used to solve a subproblem are most times only interested in finding the best columns. This means that many columns having negative reduced costs are left out because the algorithm finds "better" columns. This may be desirable in the single objective case. In the bi-objective case, however, a column which may not be so good for a subproblem may be the best for another subproblem so we are interested in heuristics that can efficiently search for these columns by modifying the ones found by a subproblem algorithm. We denote an approach which incorporate such heuristics as "Improved Point-by-Point Search (IPPS)". IPPS can be useful as a column generation based heuristic since at each iteration it tries to generate a set of columns that are relevant for several subproblems. IPPS is summarized in Algorithm 2 and the heuristic used in Step 7 obviously depends on the problem at hand.

## 3 Application Problems

In this section, we apply the ideas presented so far to compute lower and upper bounds for two BOVRPMMO. Just as for most VRPs, the subproblem encountered in both examples is

**Algorithm 2** Improved Point-by-Point Search (IPPS)

```
1: Set $\varepsilon \leftarrow \infty$, and $lb \leftarrow \emptyset$.
2: while LRMP is feasible do
3:     Solve LRMP once to obtain a vector of dual values.
4:     Let $c^*$ be the optimum, $\lambda^*$ the optimal vector, and compute $\Gamma_{\max} = \max_{k\in\bar{\Omega}} \sigma_k \lambda_k^*$.
5:     Solve the subproblem $S(\varepsilon)$ and let $\Lambda$ be the set of columns obtained.
6:     if $|\Lambda| \neq 0$ then
7:         For each column in $\Lambda$ use heuristics to generate other relevant columns from it.
8:     else
9:         $lb \leftarrow lb \cup \{(c^*, \Gamma_{\max})\}$.
10:        Set $\lambda_k \leftarrow 0$ for all $k$ such that $\sigma_k \geq \sigma^*$, and $\varepsilon \leftarrow \Gamma_{\max} - 1$.
11:    end if
12: end while
```

an elementary shortest part problem with resource constraints (ESPPRC) which we solve by the decremental state space relaxation algorithm (DSSR) [1, 13]. For each considered problem, we discuss the specific implementation of DSSR as well as the heuristic used in implementing IPPS. A complete description of DSSR can be obtained from the two references. We will also be using the same notations introduced earlier and so only their specific meanings for each example will be mentioned.

## 3.1 The Bi-Objective Uncapacitated Vehicle Routing Problem

The bi-objective uncapacitated VRP (BOUVRP) is defined on an undirected graph $G = (V, E)$ where $V = \{v_0, \dots, v_n\}$ is a set of nodes and $E = \{(v_i, v_j) : v_i, v_j \in V, i \neq j\}$ is a set of edges. Node $v_0$ is the depot where all routes should start and end. The other nodes represent $n$ customers with each having a fixed demand. A distance matrix $D = (d_{ij})$ which satisfies the triangle inequality is defined on $E$. The problem is to design a set of routes with the objectives of minimizing both the total length of all routes and the maximum total demand of customers served by any single route. The demand of each customer is to be met by at least one visiting route and the number of available vehicles as well as the capacity of a vehicle are unlimited.

### Master Problem and Subproblem

Following the notations used in Section 2, we let $\Omega$ be the set of all feasible columns. A column $k \in \Omega$ is a Hamiltonian cycle on a subset of $V$ which includes the depot. For each column $k$, $c_k$ is its length and $\sigma_k$ is the sum of the demands of the customers visited by the route it represents. By redefining the feasibility of a column to take into account the total demand of the customers visited by the route it represents, we obtain a new set of feasible columns $\bar{\Omega}$. The master problem (MP) then becomes a capacitated VRP given by:

$$\text{Minimize} \sum_{k\in\bar{\Omega}} c_k \lambda_k \tag{10}$$

$$\text{subject to} \quad \sum_{k\in\bar{\Omega}} a_{ik}\lambda_k \geq 1 \qquad (v_i \in V\backslash\{v_0\}), \tag{11}$$

$$\lambda_k \in \{0, 1\} \qquad (k \in \bar{\Omega}), \tag{12}$$

where $a_{ik} = 1$ if the route of column $k$ visits customer $v_i$. As explained in the previous section, the second objective to minimize the total demand of customers served by a single route ($\Gamma_{\max}$) does not appear in the master problem. The subproblem is given by:

$$S(\varepsilon) = \min_{k \in \bar{\Omega} \setminus \bar{\Omega}_1} \left\{ c_k - \sum_{v_i \in V \setminus \{v_0\}} \pi_i a_{ik} : \sigma_k \leq \varepsilon \right\}, \tag{13}$$

where $\pi_i$ are dual values and $\varepsilon$ is a limit imposed on the sum of the demands of customers visited by the same route.

### Subproblem Algorithm

In (13), we are supposed to find feasible routes with negative reduced costs such that the sum of demands of the subset of customers it visits is at most $\varepsilon$. The reduced cost of a route is given by its length minus the sum of the profits (dual values) associated to the nodes it visits. The only considered resource when implementing DSSR is the sum of the demands a route visits. This sum cannot be more than $\varepsilon$. For two labels $l_1$ and $l_2$ arriving at the same node, $l_1$ dominates $l_2$ if $l_1$ has a smaller reduced cost and smaller sum of demands than $l_2$. In this situation, $l_2$ is rejected.

### IPPS Heuristic

Although DSSR is able to return several columns with negative reduced cost, some columns are not generated since they are dominated by other generated ones. Nevertheless, some of these rejected columns have negative reduced costs and may be relevant for a subproblem corresponding to a different value of $\varepsilon$. The purpose of the heuristic is to find these kind of routes by using the following idea. Given a column with negative reduced cost, we try to remove a visited node from (or insert a non-visited node in) the route in such a way that we obtain a new route which is still of negative reduce cost but has different values of $c_k$ and $\sigma_k$. Due to the change in the value of $\sigma_k$, a new route found in this way can be valid for a different subproblem for which the original route was not valid. This new route may also be of negative reduced cost for a different subproblem.

## 3.2 The Bi-Objective Multi-Vehicle Covering Tour Problem

The bi-objective multi-vehicle covering tour problem (BOMCTP) is an extension of the covering tour problem (CTP) [7]. The CTP consists in designing a route over a subset of locations with the aim of minimizing the length of the route. In addition, each location not visited by the route should lie within a fixed radius of a visited location. The fixed radius is called the *cover distance*. A generic application of the CTP is given in the design of bi-level transportation networks where the aim is to construct a primary route such that all points that are not on it can easily reach it [2]. Other applications are the post box location problem [11] and in the delivery of medical services to villages in developing countries [2, 9]. A bi-objective generalization [10] as well as a multi-vehicle extension [8] of the CTP have been proposed. In the bi-objective version, the cover distance is not fixed in advance but rather induced by the constructed route. It is computed by assigning each non-visited location to the closest visited location and calculating the maximum of these distances. The objectives are to minimize the length of the route as well as the *induced cover distance*. In the multi-vehicle version, the combined length of a set of routes is minimized for a fixed cover distance. The number of locations that a single route can visit is limited by a predetermined constant $p$.

The BOMCTP discussed here, is a combination of the bi-objective and the multi-vehicle extensions of the CTP and it is defined on an undirected graph $G = (V \cup W, E)$. Set $V$ represents locations which can be visited by a route whereas the members of $W$ are to be assigned to visited locations of $V$. Node $v_0 \in V$ is the depot where all routes must start and also end. Set $E$ consist of edges connecting all pairs of nodes in $V \cup W$ and a distance matrix $D = (d_{ij})$ satisfying the triangle inequality is defined on this set. The problem consists in minimizing both the total length of a set of routes constructed over a subset of $V$ and the induced cover distance.

### Master Problem and Subproblem

Let $\Omega$ represent the set of all feasible columns. A feasible column $k \in \Omega$ is defined as a route $R_k$ which is a Hamiltonian cycle on a subset of $V$, includes the depot and visits not more than $p$ nodes. The length of $R_k$ is denoted $c_k$. For each route $R_k$, we choose a subset $\Psi_k \subseteq W$ of nodes it may cover and define $\sigma_k$ as the maximum distance between a node of $\Psi_k$ and the closest node of $R_k$. The constant $a_{ik} = 1$ if $w_i \in \Psi_k$ and $a_{ik} = 0$ otherwise. Let $\Gamma_{\max}$ represent the cover distance induced by a set of routes. Just as before, we define a new set of feasible columns $\bar{\Omega}$ where the feasibility of a column $k \in \bar{\Omega}$ depends not just on $R_k$ but also on $\sigma_k$. The master problem (MP) is given by:

$$\text{Minimize} \sum_{k \in \bar{\Omega}} c_k \lambda_k \tag{14}$$

$$\text{subject to} \quad \sum_{k \in \bar{\Omega}} a_{ik} \lambda_k \geq 1 \qquad (w_i \in W)\ , \tag{15}$$

$$\lambda_k \in \{0,1\} \quad (k \in \bar{\Omega})\ . \tag{16}$$

The function (14) minimizes the total length of the set of routes and (15) ensures that each node of $W$ is covered by a selected route. The second objective to minimize $\Gamma_{\max}$ does not appear in the above formulation for the same reasons as before. If $\pi_i$ are the dual values associated with (15) then the subproblem is

$$S(\varepsilon) = \min_{k \in \bar{\Omega} \setminus \bar{\Omega}_1} \left\{ c_k - \sum_{w_i \in W} \pi_i a_{ik} : \sigma_k \leq \varepsilon \right\}\ . \tag{17}$$

### Subproblem Algorithm

Given that $R_k \subseteq V$ whereas $\Psi_k \subseteq W$, we need to construct a route on a subset of $V$ with the aim of minimizing its length $c_k$ and also choose a subset of $W$ with the aim of maximizing the profits ($\pi_i$) associated to its members. The profit associated to a node of $w_i \in W$ can be collected at most once on any single route even though different nodes of the route may be able to cover $w_i$. Two resources are considered in implementing DSSR for this problem. The first is concerned with the number of nodes a route visits which is limited to a maximum of $p$. The second resource constraint is that a route may only cover nodes of $W$ that lie within a radius of $\varepsilon$ from a node of $V$ it visits. During the extension of a label, nodes of $W$ not yet covered by the label but which can be covered are identified and the resulting profit is subtracted from the current reduced cost of the label. Doing so ensures that we obtain the minimum possible reduced cost for each label and this is the goal of the subproblem. Checking whether a label $l_1$ dominates another label $l_2$ follows the usual rules when comparing the consumption of resources. When comparing the reduced costs, however, a factor $F_{12}$ which represents the sum of the profits associated to nodes of $W$ covered by $l_1$

but not yet covered by $l_2$ should be subtracted from the reduced cost of $l_2$. This is to ensure that no label that can lead to an optimal solution is eliminated. Similar dominance rules used in dynamic programming algorithms for solving shortest path problems like the one encountered here (called non-additive shortest path problems) are discussed in [12].

### IPPS Heuristic

The subproblem constructs a column $k$ by taking $\Psi_k$ to be all the nodes of $W$ that can be covered by the constructed route $R_k$. This helps with the goal of minimizing the reduced cost. We note, however, that $\Psi_k$ does not necessarily need to include all the nodes of $W$ that can be covered by $R_k$. Indeed, $\Psi_k$ can be chosen to be any subset of $W$ such that the sum of the associated profits exceeds the cost $c_k$. A column defined in this way is never found by DSSR for the current subproblem since it is dominated by another column defined by the same route, but covers some more nodes of $W$. The IPPS heuristic employed here relies on this observation. For each column $k$ given by $(R_k, \Psi_k)$ that is found by DSSR, we successively remove the node of $\Psi_k$ that induces the value of $\sigma_k$ (i.e. which is farthest from the closest node of $R_k$) in order to create another column $k'$ with $c'_k = c_k$ but $\sigma'_k < \sigma_k$. We recall that $\sigma_k = \max\{d_{ij} : v_i \in R_k \text{ and } w_j \in \Psi_k\}$. A column found in this way can be valid (and possibly have a negative reduced cost) for another subproblem for which the original column returned by DSSR is not valid. This is due to the different value of $\varepsilon$ each subproblem is based on.

## 4 Computational Results

### Evaluating the Quality of Lower and Upper Bounds

In order to evaluate the quality of the computed lower and upper bound sets, we used a distance based measure ($\mu_1$), and an area based measure ($\mu_2$) which were presented in [6]. Combining an area based measure with a distance based measure gives a better indication of the quality of the bounds. Roughly speaking, $\mu_1$ represents the worst distance (with respect to the range of objective values) between a point of the upper bound and a point of lower bound closest to it. Also, $\mu_2$ represents the fraction of the area that is dominated by the lower bound but not by the upper bound. This is, the area where additional points of $\mathcal{Y}_N$ can be found. If a lower bound and a corresponding upper bound are good then we expect that both $\mu_1$ and $\mu_2$ will be small in value. The smaller both values are, the better the quality of the bounds. These two measures complement each other so the quality of the bounds cannot be said to be very good if just one of the measures is small in value but the other is very big. As explained by the author, these measures can be seen to play a role analogous to the optimality gap in single objective optimization. The reader is referred to the relevant paper [6] for further explanation of the measures.

### Experiments

Experiments were conducted to evaluate the quality of lower and upper bounds obtained from the model presented (by redefining the feasibility of a column) with respect to a standard $\varepsilon$-constraint model (by explicitly adding constraints on objectives in the master problem). We recall that in the standard $\varepsilon$-constraint model, constraints that limit the value of $\sigma_k$ for a column are directly added to the master problem and nothing special is done in the subproblem. On the other hand, the new model does not explicitly add constraints to the master problem but rather limit the value of $\sigma_k$ for a column by redefining the meaning

**Table 1** Comparison of the quality of bound sets for the BOUVRP.

| | Standard | | | | | PPS | | | IPPS | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Instance | $\lvert lb\rvert$ | $\lvert ub\rvert$ | $\mu_1\%$ | $\mu_2\%$ | $\lvert lb^*\rvert$ | $\lvert ub\rvert$ | $\mu_1\%$ | $\mu_2\%$ | $\lvert ub\rvert$ | $\mu_1\%$ | $\mu_2\%$ |
| eil7 | 2 | 6 | 2.60 | 37.32 | 6 | 6 | 0.00 | 3.26 | 6 | 0.00 | 3.26 |
| eil13 | 2 | 32 | 2.92 | 25.22 | 99 | 33 | 0.13 | 2.72 | 34 | 0.10 | 2.67 |
| eil22 | 11 | 36 | 1.17 | 11.68 | 100 | 41 | 0.15 | 2.24 | 40 | 0.11 | 2.11 |
| eil23 | 2 | 11 | 1.48 | 50.14 | 130 | 14 | 0.30 | 16.24 | 17 | 0.13 | 15.90 |
| eil30 | 5 | 18 | 9.12 | 24.30 | 152 | 19 | 1.08 | 5.32 | 22 | 0.98 | 4.89 |
| eil31 | 13 | 44 | 5.30 | 10.95 | 173 | 41 | 0.88 | 2.69 | 42 | 0.40 | 2.65 |

of a feasible column both in the master problem and subproblem. The principle used to determine the values of $\varepsilon$ is the same for both models. Given a value for $\Gamma_{\max}$ in an iteration, we define $\varepsilon = \Gamma_{\max} - 1$ in the next iteration. We also wanted to compare the quality of the bounds and the computational times of PPS and IPPS. Capacitated VRP instances from the TSPLIB (`http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/`) having up to 31 nodes were used for the BOUVRP. For the BOMCTP, the Mersenne Twister random number generator (`http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html`) was used to generate instances similar to those described in the literature [7, 8, 10] but which are not publicly available. The node sets were obtained by generating $|V| + |W|$ points in the $[0, 100] \times [0, 100]$ square with the depot restricted to lie in $[25, 75] \times [25, 75]$. Set $V$ is taken to be the first $|V|$ points and set $W$ is taken as the remaining points. The distance between two points is calculated as the Euclidean distance. Five instances for every combination of $|V| \in \{40, 50\}$ and $|W| \in \{2|V|, 3|V|\}$ were generated and values of $p \in \{5, 8\}$ were tested. The exact instances used for our experiments can be found at `http://homepages.laas.fr/bmsarpon/ctp_instances.zip`. All computer codes were written in C/C++ and the LRMP was solved with ILOG CPLEX 12.4. Tests were run on an Intel(R) Core(TM)2 Duo CPU E7500 @ 2.93GHz computer with a 2 GiB RAM. Summary of results for the BOUVRP are given in Tables 1 and 2 whereas those for the BOMCTP are given in Tables 3 and 4. The values for the BOMCTP are averages over the five instances as explained. The column headings of the tables have the following meaning: $|lb|$ and $|ub|$ are the cardinalities of a lower bound and upper bound set, respectively; $\mu_1\%$ and $\mu_2\%$ are the values of the quality measures multiplied by 100; *time* is the computational time in cpu seconds; *dssr* is the number of times the sub-problem was solved with DSSR; *cols* is the total number of columns generated. Since PPS and IPPS are based on the same model, the same lower bounds were obtained and this conforms to the theory of column generation which is an exact method for the LMP. The cardinality of the common lower bound is given in the column $|lb^*|$.

From the results obtained, we see that the bounds obtained by the model that redefines the feasibility of a column are significantly better than those obtained by a standard $\varepsilon$-constraint approach. This is seen by comparing the values of $\mu_1$ and $\mu_2$ for PPS and IPPS with their counterparts from "Standard". For example, in Table 1 a standard $\varepsilon$-constraint approach obtained the values $(\mu_1\% = 9.12, \mu_2\% = 24.30)$ for the instance eil30 whereas those for PPS and IPPS were $(\mu_1\% = 1.08, \mu_2\% = 5.32)$ and $(\mu_1\% = 0.98, \mu_2\% = 4.89)$, respectively. A similar thing can be seen in Table 3 for $p = 8$, $|V| = 40$, $|W| = 80$. The values for a standard $\varepsilon$-constraint approach for this instance were $(\mu_1\% = 8.38, \mu_2\% = 25.86)$ which are very huge when compared to $(\mu_1\% = 0.32, \mu_2\% = 3.04)$ for PPS and $(\mu_1\% = 0.38, \mu_2\% = 2.46)$ for IPPS.

**Table 2** Comparison of computational times for the BOUVRP.

| | Standard | | | PPS | | | IPPS | | |
|---|---|---|---|---|---|---|---|---|---|
| Instance | time | dssr | cols | time | dssr | cols | time | dssr | cols |
| eil7 | 0.1 | 15 | 24 | 0.1 | 24 | 29 | 0.1 | 13 | 51 |
| eil13 | 1.4 | 60 | 335 | 12.3 | 355 | 916 | 8.8 | 270 | 1982 |
| eil22 | 246.3 | 298 | 2295 | 428.7 | 648 | 3480 | 289.6 | 482 | 6026 |
| eil23 | 4216.7 | 421 | 3386 | 7578.5 | 1238 | 7946 | 5979.1 | 916 | 12951 |
| eil30 | 5861.2 | 541 | 2976 | 9746.2 | 1110 | 7312 | 7114.7 | 828 | 14922 |
| eil31 | 2851.0 | 368 | 2719 | 5372.4 | 957 | 4514 | 5027.3 | 804 | 7389 |

**Table 3** Comparison of the quality of bound sets for the BOMCTP.

| | | | Standard | | | | | PPS | | | IPPS | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $p$ | $\|V\|$ | $\|W\|$ | $\|lb\|$ | $\|ub\|$ | $\mu_1\%$ | $\mu_2\%$ | $\|lb^*\|$ | $\|ub\|$ | $\mu_1\%$ | $\mu_2\%$ | $\|ub\|$ | $\mu_1\%$ | $\mu_2\%$ |
| 5 | 40 | 80 | 7 | 23 | 6.41 | 23.22 | 26 | 26 | 1.07 | 7.69 | 27 | 1.01 | 6.09 |
| 5 | 40 | 120 | 8 | 24 | 4.67 | 23.35 | 27 | 28 | 0.91 | 11.92 | 29 | 0.92 | 10.33 |
| 5 | 50 | 100 | 7 | 28 | 4.74 | 21.94 | 32 | 33 | 0.70 | 9.48 | 33 | 0.68 | 7.85 |
| 5 | 50 | 150 | 10 | 24 | 4.33 | 26.44 | 30 | 30 | 1.20 | 16.67 | 30 | 1.20 | 15.41 |
| 8 | 40 | 80 | 7 | 26 | 8.38 | 25.86 | 27 | 27 | 0.32 | 3.04 | 27 | 0.38 | 2.46 |
| 8 | 40 | 120 | 8 | 28 | 6.48 | 21.70 | 29 | 30 | 0.40 | 4.50 | 30 | 0.40 | 3.70 |
| 8 | 50 | 100 | 7 | 32 | 6.13 | 21.83 | 32 | 32 | 0.32 | 3.83 | 33 | 0.31 | 3.65 |
| 8 | 50 | 150 | 9 | 30 | 5.00 | 18.70 | 30 | 30 | 0.31 | 4.30 | 30 | 0.36 | 3.66 |

It seems natural that better values for the measures are obtained when the lower and upper bound sets contain more elements. This is probably where a standard $\varepsilon$-constraint method falls short since the lower bounds it computes contain very few points in comparison to those computed by the model on which PPS and IPPS are based. Better values of $\mu_1$ and $\mu_2$ were obtained for IPPS in comparison to PPS for the tested instances of both the BOUVRP and the BOMCTP. Since the same lower bounds were computed by both approaches for any given instance, we can attribute the better values of the measures for IPPS to the quality of the upper bounds it produces. In terms of computational times, the standard approach is the fastest and this is not so surprising given the number of points it computes and the poor quality of the bounds it produces when compared to the others. When computing the lower bounds, IPPS needs to solve fewer subproblems than PPS (see values for *dssr*) and also generates significantly more columns than (see values for *cols*). The effect is that, the computational times is significantly reduced for IPPS in comparison to PPS. Finally, since the members of an upper bound set correspond to images of feasible solutions, the results mean PPS and IPPS can be used to provide very good approximations of the nondominated set $\mathcal{Y}_N$ for the class of problems considered.

## 5 Conclusions

This paper discusses the application of column generation to bi-objective VRPs in which one objective is a min-max function. An idea for formulating these problems based on a variant of the $\varepsilon$-constraint method is presented. Instead of adding constraints on an objective

**Table 4** Comparison of computational times for the BOMCTP.

| | | | Standard | | | PPS | | | IPPS | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $p$ | $\|V\|$ | $\|W\|$ | time | dssr | cols | time | dssr | cols | time | dssr | cols |
| 5 | 40 | 80 | 27.5 | 137 | 790 | 49.5 | 228 | 1597 | 40.5 | 155 | 2099 |
| 5 | 40 | 120 | 38.9 | 163 | 1027 | 126.8 | 330 | 2571 | 94.0 | 201 | 3388 |
| 5 | 50 | 100 | 68.5 | 197 | 1240 | 205.8 | 390 | 3035 | 153.9 | 226 | 3459 |
| 5 | 50 | 150 | 42.2 | 150 | 875 | 392.5 | 486 | 4053 | 287.0 | 247 | 4054 |
| 8 | 40 | 80 | 61.0 | 217 | 1498 | 113.4 | 302 | 2283 | 103.6 | 218 | 2961 |
| 8 | 40 | 120 | 132.1 | 299 | 2205 | 511.2 | 481 | 3949 | 503.9 | 293 | 4663 |
| 8 | 50 | 100 | 281.2 | 326 | 2398 | 1343.7 | 522 | 4306 | 1012.5 | 335 | 5071 |
| 8 | 50 | 150 | 333.0 | 380 | 2872 | 1525.2 | 672 | 5799 | 1186.2 | 384 | 6005 |

in the master problem, we rather redefine the set of feasible columns to take the objective into account. We keep the strength of the model at the expense of a possibly more difficult problem. The advantages of using this model is clearly exhibited from the quality of the bounds obtained from it. We also investigate a strategy to accelerate and improve the column generation method. The proposed ideas are applied to two VRPs and the results obtained indicate that an intelligent management of columns in a multi-objective perspective can yield significant speedups in computing lower and upper bounds. Given that the time needed to compute such quality bounds can be very long, future works are aimed at finding a good compromise between the quality of bounds and the computational time. It will also be interesting to develop branching rules in order to explore the idea of a multi-objective branch-and-price algorithm.

## References

1 Natashia Boland, John Dethridge, and Irina Dumitrescu. Accelerated label setting algorithms for the elementary resource constrained shortest path problem. *Operations Research Letters*, 34(1):58–68, 2006.

2 John R. Current and David A. Schilling. The median tour and maximal covering tour problems: Formulations and heuristics. *European Journal of Operational Research*, 73(1):114–126, 1994.

3 Charles Delort and Olivier Spanjaard. Using bound sets in multiobjective optimization: Application to the biobjective binary knapsack problem. In *Experimental Algorithms*, pages 253–265. Springer, 2010.

4 Guy Desaulniers, Jacques Desrosiers, and Marius M. Solomon. Accelerating Strategies in Column Generation Methods for Vehicle Routing and Crew Scheduling Problems. In *Essays and Surveys in Metaheuristics*, volume 15 of *Operations Research/Computer Science Interfaces Series*, pages 309–324. Springer US, 2002.

5 Matthias Ehrgott. A discussion of scalarization techniques for multiple objective integer programming. *Annals of Operations Research*, 147(1):343–360, 2006.

6 Matthias Ehrgott and Xavier Gandibleux. Bound sets for biobjective combinatorial optimization problems. *Computers & Operations Research*, 34(9):2674–2694, 2007.

7 Michel Gendreau, Gilbert Laporte, and Frédéric Semet. The Covering Tour Problem. *Operations Research*, 45(4):568–576, 1997.

8 Mondher Hachicha, M John Hodgson, Gilbert Laporte, and Frédéric Semet. Heuristics for the multi-vehicle covering tour problem. *Computers & Operations Research*, 27(1):29–42, 2000.

9 M. John Hodgson, Gilbert Laporte, and Frederic Semet. A Covering Tour Model for Planning Mobile Health Care Facilities in SuhumDistrict, Ghama. *Journal of Regional Science*, 38(4):621–638, 1998.

10 Nicolas Jozefowiez, Frédéric Semet, and El-Ghazali Talbi. The bi-objective covering tour problem. *Computers & Operations Research*, 34(7):1929–1942, 2007.

11 Martine Labbé and Gilbert Laporte. *Maximizing User Convenience and Postal Service Efficiency in Post Box Location*. Cahiers du GÉRAD. Université de Montréal, Centre de recherche sur les transports, 1986.

12 Line Blander Reinhardt and David Pisinger. Multi-objective and multi-constrained non-additive shortest path problems. *Computers & Operations Research*, 38(3):605–616, 2011.

13 Giovanni Righini and Matteo Salani. New dynamic programming algorithms for the resource constrained elementary shortest path problem. *Networks*, 51(3):155–170, 2008.

14 Francis Sourd and Olivier Spanjaard. A multiobjective branch-and-bound framework: Application to the biobjective spanning tree problem. *INFORMS Journal on Computing*, 20(3):472–484, 2008.

15 Bernardo Villarreal and Mark H. Karwan. Multicriteria integer programming: A (hybrid) dynamic programming recursive approach. *Mathematical Programming*, 21(1):204–223, 1981.

# Carpooling : the 2 Synchronization Points Shortest Paths Problem *

**Arthur Bit-Monnot[1,2], Christian Artigues[1,2], Marie-José Huguet[1,3], and Marc-Olivier Killijian[1,2]**

1 **CNRS, LAAS, 7 avenue du colonel Roche, F–31400 Toulouse, France**
2 **Université de Toulouse, LAAS, F–31400 Toulouse, France**
3 **Université de Toulouse, INSA, F–31400 Toulouse, France**
`{bit-monnot, artigues, huguet, killijian}@laas.fr`

**Abstract**

Carpooling is an appropriate solution to address traffic congestion and to reduce the ecological footprint of the car use. In this paper, we address an essential problem for providing dynamic carpooling: how to compute the shortest driver's and passenger's paths. Indeed, those two paths are synchronized in the sense that they have a common subpath between two points: the location where the passenger is picked up and the one where he is dropped off the car. The passenger path may include time-dependent public transportation parts before or after the common subpath. This defines the 2 Synchronization Points Shortest Path Problem (2SPSPP). We show that the 2SPSPP has a polynomial worst-case complexity. However, despite this polynomial complexity, one needs efficient algorithms to solve it in realistic transportation networks. We focus on efficient computation of optimal itineraries for solving the 2SPSPP, i.e. determining the (optimal) pick-up and drop-off points and the two synchronized paths that minimize the total traveling time. We also define restriction areas for reasonable pick-up and drop-off points and use them to guide the algorithms using heuristics based on landmarks. Experiments are conducted on real transportation networks. The results show the efficiency of the proposed algorithms and the interest of restriction areas for pick-up or drop-off points in terms of CPU time, in addition to its application interest.

**1998 ACM Subject Classification** G.2.1 Combinatorics, G.2.2 Graph Theory, F.2.2 Nonnumerical Algorithms and Problems

**Keywords and phrases** Dynamic Carpooling, Shortest Path Problem, Synchronized Paths

**Digital Object Identifier** 10.4230/OASIcs.ATMOS.2013.150

## 1 Introduction

Due to the demographic evolution and the urban spread off during the last decades, people have moved away from urban centers and now live in residential areas. In order to decrease the urban traffic congestion and its societal issues, transport strategies have encouraged to park private cars near multimodal hubs (i.e. park and ride stations) and to use the public transport system to reach downtown destinations. However, congestion problems have moved from urban to sub-urban areas where people commute with their cars either to reach the employment areas or to connect to the public transport system. An appropriate solution, requiring little investment and reducing the ecological footprint of the car use, is

* This work was partially supported by LAAS, CNRS and ANR French national program for Security and Informatics (grant #ANR-11-INSE-010).

the promotion of shared transport, like carpooling, which enables private cars to become part of the public transport system. The main restraints of carpooling development are insecurity, payment transaction of the shared journey, low number of matches and lack of flexibility, as well as constraint feelings. For instance, regular (i.e., static) carpooling forces the driver to directly go home after work or to plan his trip in advance. Dynamic carpooling relaxes some of these constraints (few matches, lack of flexibility and constraint feelings). Dynamic carpooling should enable automatic (or semi-automatic) destination guessing and trip proposals for drivers. Regarding users, it should help real-time matching with drivers. In this paper, we address the issue of computing journeys for a driver and a passenger to carpool together in a complete trip. The two synchronized paths can be decomposed into 5 subpaths. The trip is composed of two convergent paths towards a first synchronization point, i.e. the meeting point, a shared path towards the second synchronization point, i.e., the drop-off point and two divergent paths from this drop-off point towards each destination, henceforth the name 2 Synchronization Points Shortest Path Problem (2SPSPP).

In the problem definition, we can distinguish two types of users. The *driver* drives his car and is willing to take a detour in order to pick up a passenger and drive him for some part of the trip. The *passenger* can walk or use public transportation to join a pick-up point in order to be driven. For example, as in the AMORES project[2], we can consider that the users use smartphones to communicate carpooling requests and offers, to find matches between those, and possibly to compute their optimal itineraries. In this paper, we focus explicitly on the computation of optimal itineraries for the 2SPSPP, i.e. the (optimal) pick-up and drop-off points and the 5 paths which compose the full trip as in figure 1. We consider the objective of minimizing the total travel time for both users.

## 2 Problem Statement

A multimodal transportation network is modeled with an edge-labeled graph $G = (V, E, \Sigma)$ where $V$ is the set of nodes, $\Sigma$ the set of modes (for instance foot, car or public transportation) and $E$ is the set of labeled edges. A labeled edge $(i, j, m)$ is a route from a node $i$ to a node $j$ having the mode $m$. Moreover, a cost function $c_{ijm}$ is associated to each edge $(i, j, m)$ representing the travel time. These costs may be static or time-dependent, in this case $c_{ijm}(\tau)$ gives the travel time from $i$ to $j$ in mode $m$ when leaving $i$ at time $\tau$. A path $P_{ij}$ is an ordered list of nodes from $i$ to $j$. Its cost, denoted by $len(P_{ij}, \tau)$, is the sum of the cost of each edge when leaving node $i$ at time $\tau$.

▶ **Definition 1** (2SPSPP)**.** Consider an edge-labeled graph $G = (V, E, \Sigma)$, a car driver $c$ and a pedestrian $p$ with their own origins and destinations, denoted by $o_c, d_c$ and $o_p, d_p$, and with their departure times $\tau_c$ and $\tau_p$ respectively. One aims to determine a pick-up point $x_{up}$ and a drop-off point $x_{off}$, and five paths $P_{o_p x_{up}}, P_{o_c x_{up}}, P_{x_{up} x_{off}}, P_{x_{off} d_p}, P_{x_{off} d_c}$ such that a carpooling cost is minimized.

This problem is depicted in figure 1; in this figure edges' labels represent the allowed modes in each part of the network: $\{c\}$ (ie. car) for the driver and $\{f, pt\}$ (ie. foot or public transportation) for the pedestrian.

A solution $S$ of the carpooling problem is a pair of pick-up and drop-off points $(x_{up}, x_{off})$ and five paths. The considered cost of a carpooling itinerary is the sum of travel times for the two users from their origin to their destination, i.e. difference between arrival and departure time for both users. Let us define $\tau_x^{(u)}$ the arrival time of user $u$ at point $x$, for instance $\tau_{d_p}^{(p)}$ is the arrival time of the passenger at $d_p$. For the considered overall carpooling cost, we

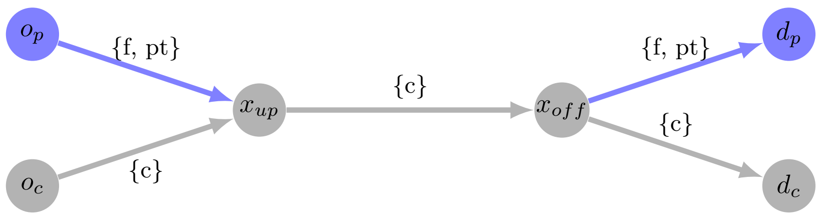


**Figure 1** Illustration of the considered carpooling problem.

point out that $\tau^{(p)}_{x_{off}} = \tau^{(d)}_{x_{off}}$ since both users arrive together at $x_{off}$, and that they leave $x_{up}$ at $\max(\tau^{(p)}_{x_{up}}, \tau^{(c)}_{x_{up}})$ since the first one arrived waits for the other.

▶ **Definition 2** (Carpooling Cost)**.** Given a solution $S$ of the 2SPSPP, we aim at minimizing $cost(S) = (\tau^{(c)}_{d_c} - \tau^{(c)}_{o_c}) + (\tau^{(p)}_{d_p} - \tau^{(p)}_{o_p})$, the total time spent traveling by both users.

$$\begin{aligned} cost(S) &= (\tau^{(c)}_{d_c} - \tau^{(c)}_{o_c}) + (\tau^{(p)}_{d_p} - \tau^{(p)}_{o_p}) \\ &= len(P_{o_p x_{up}}, \tau^{(p)}_{o_p}) + len(P_{o_c x_{up}}, \tau^{(c)}_{o_c}) + |\tau^{(c)}_{x_{up}} - \tau^{(p)}_{x_{up}}| \\ &+ 2 \times len(P_{x_{up} x_{off}}, \max(\tau^{(p)}_{x_{up}}, \tau^{(c)}_{x_{up}})) \\ &+ len(P_{x_{off} d_p}, \tau^{(p)}_{x_{off}}) + len(P_{x_{off} d_c}, \tau^{(c)}_{x_{off}}) \end{aligned} \tag{1}$$

The first line corresponds to the cost of the 2 paths $P_{o_p x_{up}}$ and $P_{o_c x_{up}}$ plus the waiting time, the second line is the cost of the path $P_{x_{up} x_{off}}$ counted twice since it is made by both users and the third one is the cost of the 2 paths $P_{x_{off} d_p}$ and $P_{x_{off} d_c}$.

We remark that we are dealing with a polynomial problem as, for fixed synchronization points, 5 calls to the Dijkstra algorithm (two of them with the time-dependent variant) are sufficient to obtain the optimal solution. As there are $O(|V|^2)$ possible synchronization points, the complexity result follows. However, a naive method of enumerating all possible pairs of synchronization points is not applicable on transportation networks having realistic size. The aim of this study is to propose an efficient algorithm for solving the 2SPSPP.

## 3 Related Work

Given a weighted graph $G = (V, E)$, an origin node $o$ and a destination node $d$, the Shortest Path Problem from $o$ to $d$ (SPP) is solved in polynomial time with the well-known Dijkstra algorithm. In this algorithm, a label $l_x = (\pi_x, p_x)$ is associated to a node $x$, where $\pi_x$ is the current cost from $o$ to $x$, and $p_x$ the reference of the predecessor node for the current best path from $o$ to $x$. A queue $Q$ is used for exploring the labels in an increasing order of their costs: the label with minimal cost is extracted from $Q$, settled and its successors are updated or inserted in $Q$. The algorithm stops when node $d$ is settled, $\pi_d$ then gives the cost of the shortest path from $o$ to $d$ and the path is obtained by exploring the predecessor $p_d$ until the origin is reached. Speed-up techniques were introduced to improve the efficiency of this algorithm for solving the one-to-one shortest path problem. In the $A^*$ goal directed search, the Dijkstra algorithm is guided towards the destination using an estimate cost between the current node and the destination $d$. The optimal solution is obtained if the estimation is a lower bound of the exact cost. In bidirectional search, two algorithms run: one from $o$ to $d$ (forward search) and one from $d$ to $o$ on the reverse graph (backward search). When a connection is found between the forward and the backward algorithms a feasible solution is obtained. However, this solution may not be optimal and the two algorithms run until there is no better solution connecting the forward and the backward labels.

In addition, different preprocessing techniques were proposed. The objective is to compute and store informations on the graph to speed-up the shortest path queries. An overview of various efficient preprocessing techniques such as landmarks, contraction hierarchy or flags is given in [4]. We only present one of them, the ALT algorithm [5] that we use later. ALT is based on landmarks and consists in computing the shortest paths from all the nodes to a (small) subset of landmarks. These precomputed shortest paths are then combined with the $A^*$ search and triangular inequality to provide strong lower bounds on the shortest paths.

Some extensions of the SPP were proposed to deal with time-dependency of travel times. When the cost function on arcs satisfies the FIFO property, the time-dependent SPP remains polynomially solvable [7] and a straightforward adaptation of the Dijkstra algorithm can be done. The FIFO property guarantees that, along any edge, it is never possible to depart later and arrive earlier. In the time-dependent DIJKSTRA algorithm, when the destination is reached, one has both the minimal cost of the shortest path and the minimal arrival time at the destination. However, many efficient techniques based on bidirectional search cannot be easily extended in the time-dependent case as the start time is given at only one node (at the origin or at the destination). For instance, an adaptation of the bidirectional ALT was proposed in [9] by considering a lower bound of travel time in the backward search. Each connection then needs to be re-evaluated to obtain the exact cost from the connection point to the destination, increasing the complexity of the problem.

When taking multimodality into account, one has to model the transportation network and the constraints on transportation modes (for instance a passenger may wish to avoid a given sequence of modes). In [3], the authors use an edge-labeled graph where a mode $m$ is associated to each edge. They propose to use a regular language $L$ to model constraints on modes and define the *regular language constrained shortest path problem* (RegLCSP). Their algorithm, called $D_{\mathsf{RegLC}}$ , is an extension of the DIJKSTRA algorithm constrained by the regular language. *Product-nodes* are simply a pair $(x, s)$ where $x$ is a node and $s$ a state in the automaton. The algorithm should be stopped as soon as a *product-node* $(d, s_f)$ is settled, where $d$ is the destination and $s_f$ is an accepting state in the automaton.

We are not aware of research addressing problems similar to the 2SPSPP. In carpooling papers, the authors usually consider variants of vehicle routing problems for solving static or long term carpooling problems (to collect several people at their home for instance and drive them at work each week or each day). Dynamic carpooling problems were also considered and several authors (see for instance [1, 10]) proposed a multi-agent architecture in which some heuristics are used to solve the matching problem between drivers and requesters. But, to the best of our knowledge, the driver is not derouted for collecting a user.

In [6], the authors propose a method for synchronizing two itineraries in a point such that the global cost of the two paths is minimized. The problem under study is the 2-Way Multi Modal Shortest Path problem in which two itineraries are defined for the same user at different times of the day between a given origin and destination. The proposed method is based on 4 multi-directional algorithms (forward and backward search) to obtain the optimal parking node such as the sum of an outgoing path and a return path is minimized. As already mentioned, the main difficulty arises when facing time-dependency, an expensive re-evaluation process is added to the 4 algorithms to obtain the exact cost of paths.

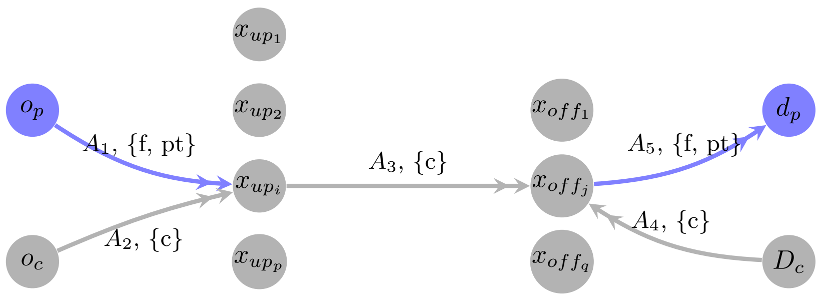


**Figure 2** General principle for solving the 2SPSPP.

## 4 The proposed approach for the 2SPSPP

### 4.1 General principle

In the problem under study, we consider that travel times for the car and foot modes are time-independent, unlike travel times for the public transportation mode that are time-dependent. Moreover, departure times are given at the origins. The proposed method aims to overcome the difficulty due to time-dependency, i.e. the use of lower bounds in algorithms where start times are unknown and the need for re-evaluation. Indeed, as public transportation is time-dependent, the use of backward search from the pedestrian's destination or from the potential drop-off points requires some (time consuming) re-evaluation. Therefore, in our method, forward search (from the origin to the destination) is used as long as it is possible to obtain the exact value of travel time and not a lower bound.

We propose a method combining 4 forward algorithms and 1 backward algorithm without any need of re-evaluation. In Figure 2, the arrows on the arcs indicates the direction of the algorithms. First, we launch 2 forward algorithms ($A_1$ and $A_2$) from the origins and 1 backward algorithm ($A_4$) from the driver's destination. Each node reached by the 2 forward algorithms $A_1$ and $A_2$ is a potential pick-up point. A forward algorithm $A_3$ is then launched from the set of potential pick-up points towards potential drop-off points. The aim of $A_3$ is to find the best origin between a set of potential origin nodes (here the pick-up points) and a set of destination node (here the drop-off points). Then, each time a node is reached by algorithm $A_3$ and the backward algorithm $A_4$, a potential drop-off point is determined. Finally, another forward algorithm $A_5$ is launched from the set of drop-off points towards the pedestrian's destination. The aim is to determine the best origin between a set of potential origin nodes (drop-off points) and a single destination node (pedestrian's destination).

Algorithms $A_1$, $A_2$ and $A_4$ are standard $D_{\mathsf{RegLC}}$ for solving the one-to-all SPP in a *multimodal* and *time-dependent* network. The multimodal constraints only state that car must be used in $A_2$, $A_3$ and $A_4$ and that either foot or public transportation can be used in $A_1$ and $A_5$. Algorithms $A_3$ and $A_5$ are dedicated to solving the best origin problem. We present in the next section how this problem can be solved.

### 4.2 The Best-Origin Problem

Given a set $S$ of several origin nodes with individual costs and arrival times (ie. $\pi_x$ and $\tau_x \forall x \in S$) and a set of target nodes $D$, we aim at selecting the best origin to minimize the cost at the destinations.

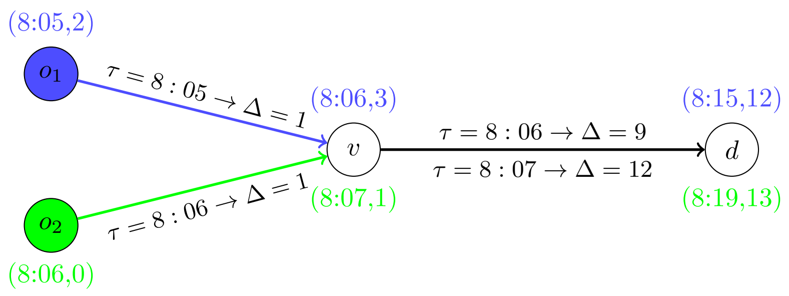

(8:05,2)
$o_1$
$\tau = 8:05 \to \Delta = 1$
(8:06,3)
$v$
$\tau = 8:06 \to \Delta = 9$
$\tau = 8:07 \to \Delta = 12$
(8:15,12)
$d$
(8:19,13)
(8:07,1)
$\tau = 8:06 \to \Delta = 1$
$o_2$
(8:06,0)

**Figure 3** An example of the Best Origin Problem with two potential origins $\{o_1, o_2\}$ and inconsistent costs and arrival times. Labels are placed above (resp. below) the node if they are issued from $o_1$ (resp. $o_2$). Edges are associated with weight $\tau = a \to \Delta$ where $\Delta$ is the cost of traversing the edge when departing at time $a$.

▶ **Definition 3** (Best Origin Problem (BOP)). Given a weighted directed graph $G = (V, E)$, a set of origins $S$ and a set of destination nodes $D$, the expected output is, for every $d \in D$, an origin $x$ having label $(\pi_x, \tau_x)$ such that, for any other origin $y \in S$ with label $(\pi_y, \tau_y)$, it holds that: $\pi_x + len(P_{xd}, \tau_x) \leq \pi_y + len(P_{yd}, \tau_y)$.

Solving BOP in time-independent networks has been done implicitly for decades using the forward Dijkstra algorithm from the origins. Each time a node is touched by the algorithm, it is updated with the best available cost. The only predecessor kept is the one providing the best cost. This problem can therefore be solved by inserting all potential origins with their initial costs into the priority queue and let the Dijkstra algorithm run until every $d \in D$ is settled. The last predecessor of $d$ in the optimal path would be the best origin.

In the time-dependent context, when there is consistency between cost and arrival time (see Definition 4), we can consider that the label with the best cost is the one with the best arrival time. Using the FIFO-property, it is easy to see that the only label we are interested in is the one with the lowest cost. The Dijkstra approach (dropping all labels with greater cost) can therefore be applied to solve this problem.

▶ **Definition 4** (Consistency between cost and arrival time). Given a shortest path solver using cost and arrival time labels, we say that cost and arrival times are *consistent* if and only if, for any two labels $(\pi_x, \tau_x)$ and $(\pi_y, \tau_y)$, $\pi_x \leq \pi_y \Leftrightarrow \tau_x \leq \tau_y$

However, the classical solution approach does not hold when costs and arrival times are not consistent. Figure 3 gives an example of the BOP with inconsistent costs and arrival times. Let $S = \{o_1, o_2\}$ be a set of origins having respective inconsistent labels $(8:05, 2)$ and $(8:06, 0)$, and a destination $d$. Travel times are time-dependent and are detailed in Figure 3. Two labels are obtained for $v$: $(8:06, 3)$ due to $o_1$ and $(8:07, 1)$ due to $o_2$. The best origin for $d$ is $o_1$ (with a cost of 12), but the best label for $v$ is the one from $o_2$. Applying the Dijkstra algorithm on this instance would discard the $(8:06, 3)$ label in $v$ and $o_2$ would be selected as the best origin, giving a suboptimal result.

To solve this problem, we propose an algorithm performing forward search only and not doing any reevaluation. This algorithm is inspired by Martins' algorithm [8] to keep track of costs and arrival times. However, we note that this algorithm stays mono-objective since we are only interested in finding the best cost in $d$. Labels are sorted by cost only and priority queues for the Dijkstra algorithm can be used. Moreover, the extension of this algorithm

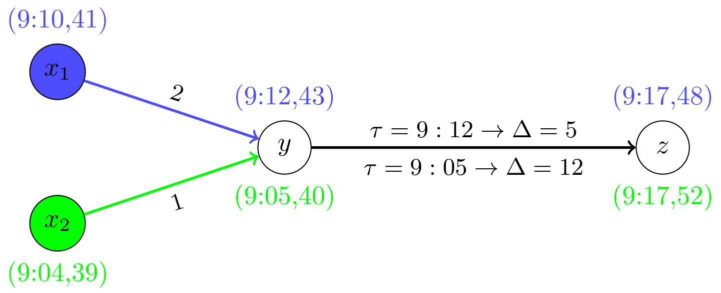

(9:10,41)
$x_1$
2
(9:12,43)
$y$
$\tau = 9:12 \rightarrow \Delta = 5$
$\tau = 9:05 \rightarrow \Delta = 12$
(9:17,48)
$z$
(9:05,40)
(9:17,52)
1
$x_2$
(9:04,39)

**Figure 4** An instance where dominance rule 2 would yield suboptimal results.

to a multimodal network is straightforward using the *product network* of the graph and the automaton representing constraints on modes.

To prune labels during the search, we introduce the following dominance rule that allows discarding labels that can not be part of an optimal solution.

▶ **Definition 5** (Exact Dominance rule (1)). Given a node $x$ and two labels $l = (\pi_x, \tau_x)$ and $l' = (\pi'_x, \tau'_x)$, we say that $l$ dominates $l'$ if and only if $\tau_x \leq \tau'_x$ and $\pi_x - \pi'_x \leq \tau_x - \tau'_x$.

▶ Proposition 6. At least one optimal solution is reachable if dominance rule 1 is applied.

**Proof.** Let us select an optimal path $P$ from an origin $o$ to $d_p$ that verifies

1. $i$ is the first node on the path such that the label $(\pi', \tau')$, extended to obtain $P$, is dominated according to rule 1 by another label $(\pi, \tau)$ of node $i$.
2. Among the optimal paths, $P$ is the one with the smallest number of arcs between $i$ and $d_p$

According to rule 1 we have $\pi \leq \pi' - (\tau' - \tau)$ and $\tau \leq \tau'$. Let $P_{id_p}$ the subpath of $P$ from $i$ to destination $d_p$. It comes: $\pi + len(P_{id_p}, \tau) \leq \pi' + len(P_{id_p}, \tau) - (\tau' - \tau)$. Since $len(P_{id_p}, \tau) \leq len(P_{id_p}, \tau') + (\tau' - \tau)$ in FIFO networks, we finally have $\pi + len(P_{id_p}, \tau) \leq \pi' + len(P_{id_p}, \tau')$. It follows that extending label $(\pi, \tau)$ from node $i$ yields an optimal path. ◀

As it will be shown in Section 6, solving the BOP with this dominance rule is not always efficient. We, then, introduce a second dominance rule which is heuristic. The pros and cons of those two dominance rules will be discussed further.

▶ **Definition 7** (Heuristic Dominance rule (2)). Given a node $x$ and two labels $l = (\pi_x, \tau_x)$ and $l' = (\pi'_x, \tau'_x)$, we say that $l$ dominates $l'$ if and only if $\tau_x \leq \tau'_x$ and $\pi_x \leq \pi'_x$.

This second dominance rule is heuristic as it may prune labels leading to optimal solutions. An illustration of this situation is given in figure 4 where nodes $x_1$ and $x_2$ are potential drop-off points, nodes $y$ and $z$ are explored by the pedestrian to reach his destination. Labels of $x_1$ and $x_2$ are extended to $y$. When considering the second dominance rule, at node $y$, the blue label $(9:12; 43)$ is dominated by the green one $(9:05; 40)$. However, the extension of the blue label to node $z$ gives a better solution $(9:17; 48)$ due to the time-dependent arc from $y$ to $z$. This models a situation where the passenger would have to wait for the same bus whether it was dropped at $x_1$ or at $x_2$. As a consequence, the second dominance rule may filter labels that could lead to optimal solutions in terms of cost since it doesn't account for those situations.

In our mono-objective variant of Martins algorithm, at first, all potential origins are inserted in a queue $Q$ with their original costs and arrival times. At each iteration, the undominated label with lowest cost in $Q$ is selected, settled and its edges are relaxed. The generated labels are inserted in $Q$. Either dominance rule can be used to prune dominated labels.

▶ Proposition 8. At each iteration, a label settled has a cost greater than or equal to the cost of any label previously settled.

**Proof.** Given that edge weights are non-negative, the cost of a label will be greater than or equal to its predecessor's. Since the priority queue selects labels with lowest cost first, all labels inserted in queue will have a cost greater or equal than the one currently selected. ◀

▶ **Corollary 9.** *In the* Mono-Objective Martins *algorithm, the lowest cost of a node is the one of its first settled label.*

Using Corollary 9, we can stop the algorithm as soon as a label is settled for all $d \in D$. By looking at predecessors, we deduce the best origins $o$ and the paths $P_{od}$.

▶ Proposition 10. There can be at most $|S|$ undominated labels per node, $|S|$ being the number of potential origins.

**Proof.** Given a potential origin $o$ with label $(\pi_o, \tau_o)$ and a node $v$, we suppose there are two paths $P$ and $P'$ from $o$ to $v$. The label generated in $v$ by following those paths would be $l_v = (\pi_o + len(P, \tau_o), \tau_o + len(P, \tau_o))$ and $l'_v = (\pi_o + len(P', \tau_o), \tau_o + len(P', \tau_o))$. Note that the second inequality of dominance rule 1 is always verified since $(\pi_o + len(P, \tau_o)) - (\pi_o + len(P', \tau_o)) = (\tau_o + len(P, \tau_o)) - (\tau_o + len(P', \tau_o))$. Thus, if $len(P, \tau_o) \leq len(P', \tau_o)$, $l_v$ dominates $l'_v$. Otherwise $l_v$ is dominated by $l'_v$. Therefore, each potential origin generates at most one undominated label per node. ◀

**Complexity.** Using Proposition 10, we deduce that there can be at most $|E| \cdot |S|$ labels inserted in $Q$. When extracted from the queue, these labels need to be checked for dominance, which can be done in $|S|$. Hence, the worst-case complexity of this algorithm is $O(|E| \cdot |S| \cdot r_Q + |E| \cdot |S| \cdot e_Q + |E| \cdot |S|^2)$ where $r_Q$ is the cost of reordering the queue after inserting one label and $e_Q$ is the complexity of extracting the next label.

We note that this worst-case complexity is greater than the one of running $|S|$ Dijkstra algorithms. However, in our experiments, these two rules allow to discard many labels.

## 5 Algorithm for the 2SPSPP

### 5.1 A sequential approach

In our method, we split the carpooling problem into three One-to-All Shortest Path Problems and two Best Origin Problems. The two BOP are using the nodes settled by the shortest path algorithms as their potential origins. We call $A_i$ the algorithm used to solve the $i^{th}$ problem and $N_i$ the set of nodes it settles. A specification of the algorithms and the problems they have to solve is given in Table 1. All five algorithms are to be executed sequentially. The 2SPSPP is solved when $d_p$ is settled by algorithm $A_5$: we are able to retrieve $x_{off}$ (best origin of $d_p$ in $A_5$) and $x_{up}$ (best origin of $x_{off}$ in $A_3$).

We saw in section 4.2, that the consistency between costs and arrival times has an impact on which algorithm can be used to solve the BOP. We will therefore study this consistency for each part of the proposed method. We call $\pi_x^{(i)}$ the cost of node $x$ in algorithm $A_i$ and $\tau_x^{(i)}$ the arrival time at $x$ for the algorithm $A_i$. We are also given $\tau_{o_p}^{(p)}$ and $\tau_{o_c}^{(c)}$, respectively the departure times of the passenger and the driver. Note that in $A_1$, $A_2$ costs and arrival

**Table 1** Specification of the algorithms used to solve the 2SPSPP for carpooling.

| Algorithm | Source | Target | Settled Nodes | Problem |
|---|---|---|---|---|
| $A_1$ | $o_p$ | All | $N_1$ | Shortest Path (forward) |
| $A_2$ | $o_c$ | All | $N_2$ | Shortest Path (forward) |
| $A_3$ | $X_{up} = N_1 \cap N_2$ | All | $N_3$ | Best Origin |
| $A_4$ | $d_c$ | All | $N_4$ | Shortest Path (backward) |
| $A_5$ | $X_{off} = N_3 \cap N_4$ | $d_p$ | $N_5$ | Best Origin |

times are consistent and then in $A_4$ we do not consider the time since it is executed on a time-independent graph and the arrival time has no impact on the rest of the method. Then, for $A_3$ and $A_5$, initial costs and arrival times of nodes in $X_{up}$ and $X_{off}$ derive from Definition 2 and are defined as follow:

$$\begin{aligned}\text{in } A_3,\ \text{for } x \in X_{up} : \tau_x^{(3)} = \max(\tau_x^{(1)}, \tau_x^{(2)});\ \text{and } \pi_x^{(3)} = \pi_x^{(2)} + \pi_x^{(1)} + |\tau_x^{(1)} - \tau_x^{(2)}| \\ \text{in } A_5,\ \text{for } x \in X_{off} : \tau_x^{(5)} = \tau_x^{(3)};\ \text{and } \pi_x^{(5)} = \pi_x^{(3)} + \pi_x^{(4)} \end{aligned} \tag{2}$$

Given this definition and recalling that cost is counted twice in $A_3$, it is fairly easy to show that, for any node $x \in N_3$, $\pi_x^{(3)} = 2 \times \tau_x^{(3)} - \tau_{o_p}^{(p)} - \tau_{o_c}^{(c)}$. Hence, costs and arrival times are consistent in $N_3$. However, breaking down the cost of a node $x \in X_{off}$ leads us to $\pi_x^{(5)} = 2 \times \tau_x^{(3)} - \tau_{o_p}^{(p)} - \tau_{o_c}^{(c)} + \pi_x^{(4)}$, showing that costs and arrival times are not consistent in $N_5$ (since $X_{off}$ is a subset of $N_5$).

According to those results, a Dijkstra like algorithm can be used for solving BOP in $A_3$. However, because of the inconsistency between costs and arrival times in $A_5$, Mono-Objective Martins has to be used to make sure no solution is discarded.

The complexity of this approach falls back on the one of four Dijkstra algorithms and one Mono-Objective Martins. Since any node of the graph can be a potential drop-off point, the worst-case complexity of our method is $O(|E| \cdot |V|^2)$ when using a binary heap.

## 5.2 Integrated Approach

The sequential approach raises the problem of exploring four times the whole graph. In this section, we present a method integrating the five algorithms to speed up the search. The idea is to have all five algorithms initialized and select the one with the lowest cost in its heap for execution as illustrated in the algorithm given in Listing 1. When a pick-up (resp. drop-off) point is discovered, it is dynamically inserted into $A_3$ (resp. $A_5$)'s heap.

**Listing 1** Integrated approach: the algorithm with lowest cost is selected for execution.

```
// Init : insert $o_p$ in $A_1$'s heap, $o_c$ is $A_2$'s heap and $d_c$ in $A_4$'s heap
while not all heaps empties
  $k$ = number of algorithm with smallest cost in heap
  // run one iteration in selected algorithm
  // and retrieve the settled node
  $x$ = Algo[$k$].make_one_iteration() // $x$ is settled in $A_k$
  if $x = d_p$ and $k = 5$ then stop // Problem solved
  if $k = 1$ or $k = 2$ then check pick-up point
  if $k = 3$ or $k = 4$ then check drop-off point
stop // no solution found
```

Initialization is done by inserting the origin of the passenger, the origin of the driver and the destination of the driver into, respectively, $A_1$, $A_2$ and $A_4$'s heaps with a zero cost and departure times from the origins.

An iteration of our method starts by selecting $k$ such that the next label to be settled in $A_k$ has the lowest cost among all algorithms' heaps. Then, $A_k$ makes one iteration (settling the next label and relaxing its edges) and yields the node $x$ it just settled. If $d_p$ was settled by $A_5$, the problem is solved. Otherwise, we check if $x$ can be used as a pick-up or drop-off point. A node $x$ is admissible as a pick-up point if it has been settled by $A_1$ and $A_2$. If that's the case, a new label $(x, \pi_x^{(3)}, \tau_x^{(3)})$ is inserted in $A_3$'s heap (computed with first line of Equation 2). A similar approach is taken for drop-off points: if $x$ was settled by $A_3$ and $A_4$, a label $(x, \pi_x^{(5)}, \tau_x^{(5)})$ is inserted in $A_5$'s heap (second line of Equation 2).

When executed on a *product network*, one has to make sure pick-up (resp. drop-off) nodes correspond to start states in the automaton modeling constraints on modes. Furthermore, they have to derive from nodes with accepting states in $A_1$ and $A_2$ (resp. $A_3$ and $A_4$).

▶ Proposition 11. In Listing 1, a label settled by the algorithm has a cost greater than or equal to the cost of any label previously settled.

**Proof.** There are two ways of inserting a label in our algorithm: when executing one step of Dijkstra or Mono-Objective Martins and when creating a new pick-up or drop-off label. In both Dijkstra and Mono-Objective Martins, no node with lower cost might appear as an effect of settling a node. Insertion of pick-up and drop-off points is done when a node $n^{(l)}$ is settled and the cost of the newly created label is always greater than $\pi_n^{(l)}$ (see the previous section for the costs expressions). Thus, every newly created label's cost will be greater or equal than the ones previously settled. Since we select the lowest label of all heaps, labels are settled by increasing cost. ◀

▶ **Corollary 12.** *(Correctness) When the node $d_p$ is settled in $A_5$, $\pi_{d_p}^{(5)}$ is the minimal carpooling cost.*

## 5.3 Restrictions on pick-up and drop-off points

A carpooling problem usually comes with preferences about where the pick-up and drop-off can occur. In this part, we present how such preferences can be used for guiding and stopping our method.

Let $Z_{up}$ be a set of nodes accessible by both the passenger and the driver. When restricting pick-up points to be in $Z_{up}$, it is easy to see that the goal of $A_1$ and $A_2$ is to settle all nodes in $Z_{up}$ and that they can stop once they have done it. This defines a stop-condition.

Furthermore, we would like to guide $A_1$ and $A_2$ towards $Z_{up}$. Suppose we have a set of consistent heuristic $h_t(u)$ that gives a lower bound of the distance from $u$ to $t$. To guide towards an area $Z$, we define $H_Z(u) = \min_{z \in Z} h_z(u)$. Combining consistent heuristics with min results in a consistent heuristic. Furthermore, $\forall z \in Z : H_Z(z) \leq 0$. Hence, $H_Z(u)$ can be used in the $A^*$ algorithm for guiding towards a set of nodes $Z$. In practice, using this heuristic results in guiding towards the closest node of the area.

However, this raises the problem of computing $|Z|$ heuristics at every iteration of the algorithm. We note as $d(u, v)$ the length of the shortest path from $u$ to $v$. For every landmark $L$ and every node $t$, algorithm ALT [5] provides us with two consistent heuristics: $h_t^+(u) = d(u, L) - d(t, L)$ and $h_t^-(u) = d(L, t) - d(L, u)$. Taking the minimum of each of those functions leads us to $H_Z^+(u) = d(u, L) - \max_{z \in Z} d(z, L)$ and $H_Z^-(u) = \min_{z \in Z} d(L, z) - d(L, u)$. Note that $\max_{z \in Z} d(z, L)$ and $\min_{z \in Z} d(L, z)$ are not dependent on $u$ and are to be computed only once per carpooling problem. The final heuristic we propose to use is given by taking the max of $H_Z^+$ and $H_Z^-$ over all landmarks.

We can use this heuristic in $A_1$ and $A_2$ to guide towards $Z_{up}$. A similar approach can be taken when we are given a set $Z_{off}$ of potential drop-off points to (a) stop $A_3$ and $A_4$ once they have settled all potential drop-off points (b) guide $A_3$ and $A_4$ towards $Z_{off}$.

## 6 Experimental study and discussion

All experiments were conducted under Ubuntu 13.04 on an HP Pavilion g6 with 4GB of RAM. The processor is a 2.10GHz Pentium-4 with 2MB of L2 cache. Algorithms are implemented in C++ and compiled with gcc with optimization level 2. The source code is available as free software under a *CeCILL-B* license[1]. We use a multi-modal graph modeling the French regions of Aquitaine and Midi-Pyrénées. All transportation data used in these experiments are free data. Road network corresponds to the OpenStreetMap[2] datasets and were provided by GeoFabrik[3]. Our public transportation network is based on The General Transit Feed Specification[4] format. When converted into an edge-labeled multi-modal graph, it contains 629 765 nodes (21 439 of them being public transportation nodes) nodes and about 5 millions edges (edges are duplicated for every transportation mode).

We consider 3 cities to define our instances: Toulouse, Albi and Bordeaux[5]. Both users start their journey in Bordeaux, the passenger is willing to go to Toulouse and the driver goes to Albi. Origins and destinations are randomly chosen in the respective cities and the start times of both users are identical during daytime (to have access to public transportation). In this configuration, passenger and driver typically meet in Bordeaux. The passenger is dropped-off in Toulouse and the driver continues his journey towards Albi. All presented results are an average over 50 of those instances using the presented integrated approach.

To measure the efficiency of stop conditions and guiding, we use two different restrictions on pick-up and drop-off points:

- *Cities*: $Z_{up}$ (resp. $Z_{off}$) contains all car accessible nodes within 20 minutes walk from Bordeaux (resp. Toulouse)'s public transports. Those areas contain respectively 29 865 and 46 584 nodes. In practice, these correspond to the whole cities.
- *10-min*: $Z_{up}$ (resp. $Z_{off}$) contains all car accessible nodes within 10 minutes by foot or public transportation from $o_p$ (resp. to $d_p$). Areas defined this way contain, on average, a few hundred nodes.

The three tested configurations are (a) *unrestricted*: the integrated approach defined in Section 5.2, (b) *stop-conditions*: stop conditions based on the areas *Cities* or *10-min* (c) *stop-conditions-guided*: stop conditions and landmarks in $A_2$, $A_3$, and $A_4$ to guide towards the pick-up and drop-off areas.

Tables 2 and 3 give the results of our method using respectively dominance rule 1 and dominance rule 2. In the first column, we give the tested configuration. The second, third, fourth and fifth columns present the runtime in ms, the number of settled labels, the number of labels per node in $A_5$ (for solving the Best Origin Problem) and the average carpooling cost over the 50 instances. There is a significant gap between the two dominance rules,

[1] `http://projects.laas.fr/MuPaRo/`
[2] `http://www.openstreetmap.org/`
[3] `http://www.geofabrik.de/`
[4] `https://developers.google.com/transit/gtfs/`
[5] Bordeaux-Toulouse is a two hours and a half drive while Toulouse-Albi takes about one hour.

**Table 2** Results with dominance rule 1 (exact).

| Restrictions | Runtime (ms) | Settled labels | Labels/node in $A_5$ | Cost (s) |
|---|---|---|---|---|
| – | 48 377 | 5 610 354 | 21.52 | 24 607 |
| cities | 6 212 | 1 135 823 | 13.99 | 24 610 |
| cities-guided | 5 910 | 928 487 | 13.99 | 24 610 |
| 10-min | 603 | 374 974 | 4.54 | 24 881 |
| 10-min-guided | 220 | 122 706 | 4.54 | 24 881 |

**Table 3** Results with dominance rule 2 (heuristic).

| Restrictions | Runtime (ms) | Settled labels | Labels/node in $A_5$ | Cost (s) |
|---|---|---|---|---|
| – | 4 316 | 1 793 205 | 1.17 | 24 621 |
| cities | 1 139 | 585 760 | 1.26 | 24 623 |
| cities-guided | 853 | 378 404 | 1.26 | 24 623 |
| 10-min | 571 | 372395 | 1.15 | 24 881 |
| 10-min-guided | 195 | 120 126 | 1.15 | 24 881 |

especially without restrictions on pick-up and drop-off areas. The average runtime with the exact dominance rule without restriction is about 48 sec. and goes down to 4.3 sec. with the heuristic rule, for a cost increasing of only 14 seconds. This gap comes from algorithm $A_5$ that has a high number of labels per nodes. With restrictions, this gap is shortened and the two rules are very close for the *10-min* restriction with guided heuristic as they provide the same carpooling cost with a similar runtime. With restrictions, our algorithm has acceptable runtime with the two dominance rules. Moreover, one can see the interest of the heuristic dominance rule that leads to a small number of labels per node in $A_5$ giving a runtime performance close to DIJKSTRA's on an equivalent BOP with consistency. In our instances, the two dominance rules discard many labels as, in the unrestricted configuration, there is on average 366 745 drop-off points evaluated as potential origins in the BOP.

In these tables, the stop conditions yield a major improvement. The gain of guiding our algorithms is much more noticeable for the *10-min* restriction than for the *Cities* restriction. This difference is mainly due to the quality of our guidance-heuristic increasing with smaller areas. For the *Cities* restriction, carpooling solutions are mainly identical to solutions for the unrestricted variant. When considering the *10-min* restriction, the cost of carpooling solutions is increasing of 259 seconds comparatively to the unrestricted variant.

Table 4 and 5 give the average cost[6] and, in brackets, number of nodes settled by each algorithm of our method for each dominance rule. Firstly, one can see the interest of the integrated approach comparatively to the sequential one on $A_1$, whose exploration is limited to a small part of the network. It is not the case for $A_2$ and $A_4$, they both settle all nodes because the use of the car allows exploring the graph while keeping the cost low. The benefits of using the integrated approach increases with the size of the network.

Secondly, these tables show that, in terms of number of settled nodes, restrictions have an impact on all algorithms but this impact is more important for $A_2$, $A_4$ and $A_5$. Algorithms $A_2$ and $A_4$ only consider the car mode and can, when there is no restrictions, explore the whole graph with low cost before a solution is found. Moreover, in terms of number of settled labels, the guiding variant has a light impact on $A_2$ but a large impact on $A_3$ and $A_4$ since considered paths are longer.

[6] Recall that the cost in algorithm $A_3$ is counted twice

**Table 4** Dominance rule 1 (exact): Average Cost and Number of labels settled by each algorithm. The waiting times are respectively 97, 103, 103, 439, 439.

| Restrictions | $A_1$ | $A_2$ | $A_3$ | $A_4$ | $A_5$ |
|---|---|---|---|---|---|
| - | 830 (70023) | 896 (569033) | 9456 (366412) | 3666 (569024) | 204 (4035862) |
| cities | 824 (45120) | 897 (57213) | 9457 (318811) | 3666 (119432) | 203 (595247) |
| cities-guided | 824 (45120) | 897 (52311) | 9457 (146338) | 3666 (89443) | 203 (595275) |
| 10-min | 492 (252) | 930 (17977) | 9619 (275551) | 3727 (77980) | 53 (3214) |
| 10-min-guided | 492 (252) | 930 (10548) | 9619 (68141) | 3727 (40551) | 53 (3214) |

**Table 5** Dominance rule 2 (heuristic): Average Cost and Number of labels settled by each algorithm. The waiting times are respectively 97, 103, 103, 439, 439.

| Restrictions | $A_1$ | $A_2$ | $A_3$ | $A_4$ | $A_5$ |
|---|---|---|---|---|---|
| - | 830 (70048) | 896 (569033) | 9478 (366754) | 3689 (569024) | 150 (218346) |
| cities | 824 (45120) | 897 (57213) | 9479 (318811) | 3689 (119432) | 149 (45184) |
| cities-guided | 824 (45120) | 897 (52311) | 9479 (146338) | 3689 (89443) | 149 (45192) |
| 10-min | 492 (252) | 930 (17977) | 9619 (275551) | 3727 (77980) | 53 (635) |
| 10-min-guided | 492 (252) | 930 (10548) | 9619 (68141) | 3727 (40551) | 53 (634) |

It should be noted that the optimal drop-off point is the passenger's destination in 29 instances (over 50) in all configurations. This leads to the average cost in $A_5$ being small. However the passenger's origin is never selected as the pick-up point since any waiting time is considered as part of the cost. As expected, restrictions limit the cost of the passenger's trips, this cost being transfered on waiting time and driver's costs.

## 7 Conclusions

In this paper, we propose a new algorithm to efficiently solve the 2SPSPP problem aiming at computing two synchronized paths for a driver and a pedestrian in a carpooling application, while minimizing the total travel time. Obtaining a solution is a matter of seconds on a large regional network. We also study the Best Origin Problem and exhibit precise conditions for which the problem can be challenging and benefits from a multi-label algorithm. For this problem, we propose exact and heuristic dominance rules.

Furthermore, it is worth noting that our approach of splitting the 2SPSPP into several Shortest Path and Best Origin Problems is very flexible and can easily be used to solve related problems. For instance, to solve two subproblems of the 2SPSPP: where the two users have the same origin or the same destination. Moreover, our approach is flexible enough so that other carpooling costs can be considered as long as consistency between costs and arrival times is preserved. But, one should notice than, even if the consistency is not preserved, the proposed method can be adapted by running our mono-objective variant of Martins algorithm for the best origin subproblems, leading to a more time-consuming approach.

We propose to use restricted drop-off and pick-up areas and we introduce a heuristic based on landmarks to guide the search towards these areas. These restrictions allow to obtain good solutions in less than one second, taking advantage of —highly desirable in practice— user-defined pick-up and drop-off areas with very low impact on optimality.

Future research directions include a better definition of restriction areas and integration of other acceleration techniques such as contraction hierarchies to speed up the algorithm.

We suppose in this paper that a matching was done (usually based on geographical information) between a driver and a pedestrian. Nevertheless, our method can be included to improve the matching. While we only consider two agents, our approach may be extended to a few number of pedestrians and one driver, either by using the same pick-up and drop-off points or by enumerating the set of pick-up and drop-off points. In addition, extension to a greater number of pedestrians may introduce an higher level of complexity by increasing the number of pick-up and drop-off points and by the need of re-evaluation of some paths from drop-off points to pedestrians' destinations in the time-dependent part of the network. Further studies may then be conducted to evaluate the computational time of such approaches.

In the 2SPSPP, we consider the minimization of a carpooling cost representing the total travel time of the two itineraries for pedestrian and driver. This carpooling cost introduces a cooperation between the two agents who both aim at reducing the total cost. However, it would be interesting to consider other objectives such as cost or profit for pedestrians and drivers in addition to the cost based on travel time. The proposed method can be seen as a first step towards a multi-objective approach or a fair optimization combining combinatorial optimization and game theory.

**Acknowledgements** We would like to thank Dominik Kirchler for his precious comments on the Best Origin Problem and on the dominance rule.

## References

1 G. Arnould, D. Khadraoui, M. Armendáriz, J. C. Burguillo, and A. Peleteiro. A transport based clearing system for dynamic carpooling business services. In *11th International IEEE Conference on ITS Telecommunications (ITST)*, pages 527–533, 2011.

2 C. Artigues, Y. Deswarte, J. Guiochet, M.-J. Huguet, Marc-Olivier Killijian, D. Powell, M. Roy, C. Bidan, N. Prigent, E. Anceaume, S. Gambs, G. Guette, M. Hurfin, and F. Schettini. Amores: an architecture for mobiquitous resilient systems. In *Proceedings of AppRoaches to MObiquitous Resilience (ARMOR'12), a EDCC workshop.*, 2012.

3 C. L. Barrett, R. Jacob, and M. Marathe. Formal-Language-Constrained Path Problems. *SIAM Journal on Computing*, 30(3):809–837, 2000.

4 D. Delling, P. Sanders, D. Schultes, and D. Wagner. Engineering route planning algorithms. In *Algorithmics of Large and Complex Networks*, volume 5515 of *LNCS*, pages 117–139, 2009.

5 A. V. Goldberg and R. F. Werneck. Computing point-to-point shortest paths from external memory. In *ALENEX/ANALCO*, pages 26–40. SIAM, 2005.

6 M.-J. Huguet, D. Kirchler, P. Parent, and R. Wolfler Calvo. Efficient algorithms for the 2-Way Multi-Modal Shortest Path Problems. In *International Network Optimization Conference (INOC)*, 2013.

7 E. Kaufman and R. L. Smith. Fastest paths in time-dependent networks for intelligent vehicle-highway systems applications. *IVHS Journal*, 1(1):1–11, 1993.

8 E. Martins. On a multicriteria shortest path problem. *European Journal of Operational Research*, 16(2):236–245, 1984.

9 G. Nannicini, D. Delling, D. Schultes, and L. Liberti. Bidirectional A* search on time-dependent road networks. *Networks*, 59(2):240–251, 2012.

10 M. Sghaier, H. Zgaya, S. Hammadi, and C. Tahon. A novel approach based on a distributed dynamic graph modeling set up over a subdivision process to deal with distributed optimized real time carpooling requests. In *14th International IEEE Conference on Intelligent Transportation Systems (ITSC)*, pages 1311–1316, 2011.